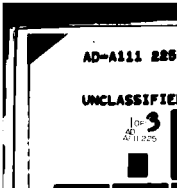
AD-A111 225 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
SURVIVABLE AVIONICS COMPUTER SYSTEM.(U)
NOV 80 P R MONSON, C A MONSON, M C PEASE F33615-80-C-1014
UNCLASSIFIED NL



SRI International

(2)

Final Report

November 1980

SURVIVABLE AVIONICS COMPUTER SYSTEM

by. P. R. MONSON C. A. MONSON M. C. PEASE C. E. WISCHMEYER

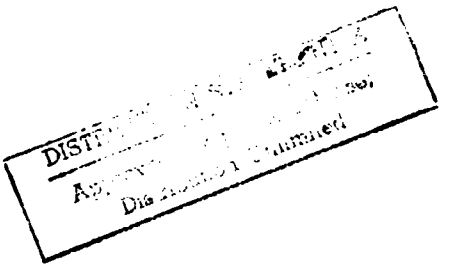
Prepared for:

COMMANDER
U.S. AIR FORCE WRIGHT AVIONICS LABORATORY
WRIGHT-PATTERSON AIR FORCE BASE
DAYTON, *OHIO 45433*

Attention: CAPT. T. HALL
AFWAL/AAA-1

CONTRACT F33-615-80-C-1014

SRI Project 1314



SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: SRI INTL MPK
TWX: 910-373-1246

SRI International

Final Report *November 1980*

SURVIVABLE AVIONICS COMPUTER SYSTEM

By: P. R. MONSON C. A. MONSON M. C. PEASE C. E. WISCHMEYER

Prepared for:

COMMANDER
U.S. AIR FORCE WRIGHT AVIONICS LABORATORY
WRIGHT-PATTERSON AIR FORCE BASE
DAYTON, OHIO 45433

Attention: CAPT. T. HALL
AFWAL/AAA-1

CONTRACT F33-615-80-C-1014

SRI Project 1314

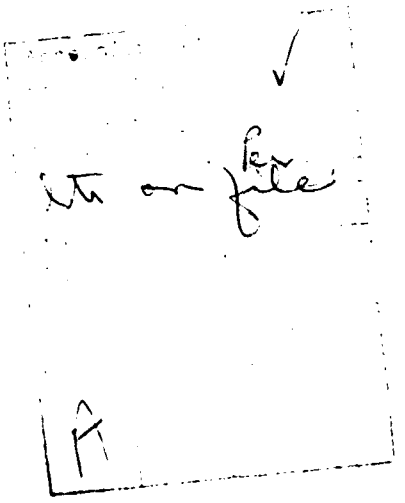
Approved by:

CHARLES J. SHOENS, *Director*
Systems Techniques Laboratory

DAVID A. JOHNSON, *Executive Director*
Technology and Systems Development Division

SRI International
333 Ravenswood Ave • Menlo Park CA 94025
415-859-6200 • TWX 910-373-2046 • Telex 334-486

CONTENTS



ILLUSTRATIONS

ILLUSTRATIONS (Continued)

ILLUSTRATIONS (Concluded)

TABLES

I INTRODUCTION

This report summarizes a seven-month effort to adapt the SRI CHAMP (Cooperative Highly Available Multiprocessor) architecture and operating system to Air Force Avionic computing needs in order to create battle-survivable computers. The CHAMP system is an extensible multiprocessor network in which the computer is made up of an assembly of communicating processors, each having its own memory and each with its own copy of the operating system. To enhance survivability in the event of hostile damage, there are no central resources and all connections to external equipment are multiple connections.

The study and simulations described in this report show that such an architecture is applicable to aircraft problems and has a high degree of potential survivability, shows by example, that existing avionics computer code can be divided into modules that are suitable for use on a distrupted computer architecture. Also included in this report is a detailed description of the operating system algorithms that are needed and an analytical proof of the valid functioning of one of these: the computing load management algorithm.

The idea of a multiprocessor is not a new one. For years computer scientists have sought to join computers together into cooperating networks to work on ever larger problems. Until recently, the size and expense of such a network was too great for use in any but the most specialized applications. One advantage of multiprocessors is that in case one or more of the units were to fail, that the others could continue to process user applications. An example of a very large multiprocessor network is the ARPA net, in which many main-frame computers are linked by a globe-spanning communication system to provide users with access to computers in various parts of the world.

Many experimental systems on a smaller scale, some of which are fairly well known, are the C.MMP and the CM* architectures of Carnegie Mellon; X-Tree, under development by the Computer Science

Department at U.C. Berkeley; and Mu Net, under development at Massachusetts Institute of Technology. All of these systems and many others not mentioned have in common the use of many small processors to provide an expanded computing capability. CHAMP differs from all of these systems in various ways, but its primarily differences are the ability to survive damage and its increased computing ability gained by cooperation among processors.

A major critisism of all such architecture has been that it is difficult to fragment a large problem, which is normally run on a sequential-style computer, into autonomous pieces small enough to run on cooperative, individual, microprocessor-based computers. A problem has been how to gain the cooperation among the modules of code without having to create a central coordinating element in the operating system.

To some extent, the answer to this question lies in the concept long used by analog computers to solve problems relating to physical systems. Analog computing elements are programmed by joining them together in cooperating networks to produce an electrical circuit that is analogous to the physical system under study. In such computers there is no central control, except that used to begin the processing. In fact, in many processes requiring the use of a computer the program structure can be visualized as a network of computing tasks functioning together in a way similar to an analog computer. All that is needed of the computing modules is a place to start and directions about other modules with whom to communicate.

The CHAMP avionics study illustrates this concept by examining, in detail two avionic functions: navigation (as embodied in the DAIS system), and flight control (as extracted from the DAIS flight control simulation algorithm). This report describes how these functions, when broken into small sub-modules whose communications with each other are well defined, can be processed in a multiprocessor environment without central control. The source listings provided by the DAIS program were in structural form and already highly modular; therefore, SRI's task was facilitated.

The additional element needed to survive damage is, of course, a method of backing-up or sparing the active computing modules. Several techniques are available for this purpose, the most elaborate of which is the Software Implemented Fault Tolerance (SIFT) developed at SRI. However, for the CHAMP network, SRI has devised a basic fault-tolerating communication and load distribution system which allows local control at each microprocessor without central intervention. Although this technique does not provide the extreme computational reliability of a SIFT-style system, it does serve as the base upon which to build a highly reliable and survivable computing system. If desired, the SIFT algorithm can be implemented on the CHAMP network to achieve SIFT-style results.

A significant breakthrough was achieved during the course of this contract--an analytical proof was constructed showing that locally controlled and locally effective load-distribution or load-leveling algorithms for multiprocessor networks are possible. This proof also shows under which specific conditions these algorithms are well behaved and lead to the desired result. Although these algorithms are not globally optimum in the sense that computational throughput is maximized, they are optimum in the sense that redistribution of the load in the event of damage is orderly and reaches a stable state quickly.

In order to better visualize this concept, consider the diagram in Figure 1. The essential problem in multiprocessor systems is to map a set of communicating software modules onto a set of communicating hardware modules without requiring any explicit knowledge on the part of the progammer of the connection and make-up of the hardware network. Figure 1 shows the linkage between the hardware modules on the lower half of the diagram and the linkage between the software modules on the upper half of the diagram with the mapping under the control of the operating system shown in the center of the diagram. This illustration is described in more detail in Section II.

Section III discusses the avionics software showing its modularity and linkages. It also describes the CHAMP network sizing design.

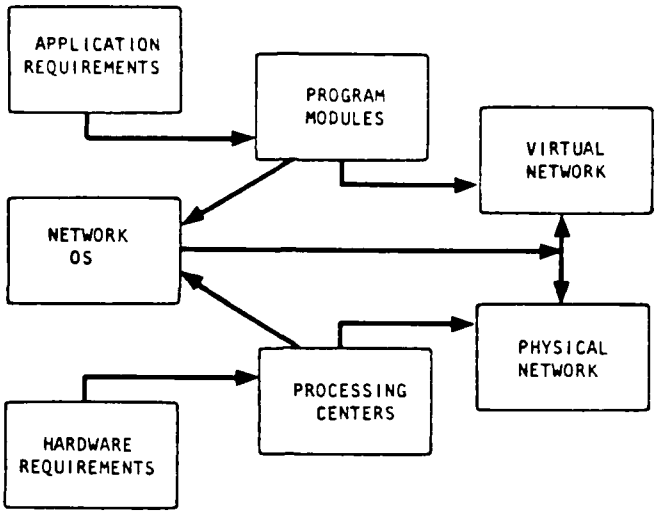


FIGURE 1 INTERFACE BETWEEN VIRTUAL AND PHYSICAL NETWORK MANAGED BY NETWORK OS

The CHAMP operating system is described in Sections IV and V. The four appendices of this report, included in the interest of making this volume a complete compendium of the work performed on this project, contain the technical descriptions of the load-balancing and communications algorithms introduced in Sections IV and V.

Appendices A and B show the structured program design for the communications and load-levelling algorithms as they would be implemented in local hardware. These algorithms are based on the results of two computer simulations written in Interlisp and run on SRI's KL-10 computer.

Appendices C and D were published as technical reports resulting from previous work on this project.

In order to develop the explanation of the Survivable Avionics Computer Study, and to understand the application of the CHAMP concept, occasional descriptions of the CHAMP process are required, and are inserted at appropriate locations throughout the text.

II BACKGROUND

CHAMP is a set of design principles and viewpoints for the development of multiprocessor architectures to emphasize certain operational characteristics. It is applicable to a variety of application environments, but does assume certain characteristics of the architecture, the application environment, and of the programming methodology, all of which are important for the following goals:

Reliability--A very high degree of fault tolerance and survivability (or durability) is a critical need for many applications.

Expandability--A system that is expandable as new resources become available, or as the demands of the application environment increase, is important in many environments.

Transportability--The ability to transport a system without reprogramming to new installations using a variety of hardware configurations may be very useful.

Maintainability--The simplification of system maintenance, and of the logistics to support the system, may be critical. For this purpose, a system should allow interchangeable use of various generations of components, incorporating improved versions as they become available, yet still be able to use old versions when new designs are unavailable.

Figure 1 diagrams the main relationships that are emphasized in the CHAMP concept as it addresses these goals.

The upper and lower halves of Figure 1 are complementary and symmetrical--the upper half outlines the development of the application program, and the lower half describes the hardware. The two halves are separated, (except for the network executive) at the boundary. Each half is invisible to the other half--the application program does not know what hardware resources are available or their connections the hardware configuration does not depend on the details of the application program. The separation of these two halves is considered essential for the goals listed above, based on the following considerations:

Reliability--The independence of the application program of the hardware configuration allows fault-tolerant actions. In particular, it allows program modules to be moved to different locations in the physical network to avoid failed components.

Expandability--The programming configuration allows making use of expanded facilities, thereby redistributing the program modules to exploit the additional capabilities. Similarly, if the scope of application requirements changes, system changes may be limited to the introduction of new program modules which are required and that are not automatically generated by some existing program module. The assignment of the new modules to processing centers will be done by the network executive without requiring or permitting intervention by the programmer or operator.

Transportability--Independence of program and hardware makes it possible to move to a new physical environment with different resources and a different configuration without reprogramming.

Maintainability and Logistics--Separation of the program from the system's hardware design allows the use of replacement processors which are not identical to those in the existing system (perhaps improved models, or perhaps older models which have not yet been replaced in the supply line) without reprogramming. The network executive is responsible for redistributing the load to make the best use of available resources.

The key to maintaining this separation is in the block called the network executive. This unit has responsibility for managing the interface between the virtual and real networks, across the boundary. It creates and maintains the map between the real and virtual networks which describes the assignment of program tasks to processing centers. It adjusts this map dynamically as faults occur or as events change the availability of the hardware or the needs of the application. The network executive uses information from the program modules regarding their requirements, from the processing centers regarding their availability, and adjusts system operations as required.

In studying what features will provide required system characteristics, we assume that the applications program can be written to exhibit certain features. Not all programs can be written in this way, and many application problems have not yet been programmed as required. However, it has been demonstrated that some important applications can be programmed in the required way, and that there are many others that are plausible candidates for further study. The main characteristics assumed of the application program are as follows:

- It is assumed that the program has been written as a set of real or virtual modules, each of which is small enough to be executable in, at most, a single processing center. The modules may be virtual in the sense that the procedures used by a module may be resident in the host processing center and available to more than one module.
- It is assumed possible to define the scope of each program module so that it is nearly automonous. This requires the module to have most of the data and values it needs for its tasks so that there is only occasional need for inter-module coordination.
- It is expected that some if not all, modules, have continuing responsibilities and impose continuing demands on the resources of the system. It also may be necessary to generate new modules on demand during operations. If so, the mechanics for doing this are provided in some permanent module(s).
- It is assumed that a useful estimate of the demands made by any module either is provided by the programmer or can be generated from the history of operations.
- It is expected that coordination among the modules can be adequately maintained by messages passed among them. These messages are not expected to require high-bandwidth capabilities. We do not exclude the possibility of signal processing operations which may require dedicated, high-bandwidth channels. If so, this is a requirement that would be handled separately from the coordination requirement that is our concern here.
- It is assumed that the essential program modules are restartable from copies of themselves using checkpoint data. This allows maintaining the copies and checkpoint data in different processing centers than those executing the module so that they will remain available for recovery after failures are detected.
- The modules are assumed to be programmed for graceful degradation when needed.

There is also an analogous set of assumptions about the hardware:

- It is assumed that the hardware is organized as a network of processing centers. We speak of processing centers rather than microcomputers since some centers may be specialized in particular functions such as I/O or signal processing computations.
- Regardless of internal differences of design or function, it is assumed that all centers provide the same interface to the network so that network integrity is not dependent on configuration.

- The processing centers are assumed to be asynchronous, each executing its tasks in its own time and requiring only occasional coordination with other centers.
- The processing centers are loosely coupled (i.e., they do not depend on shared memory for communications) but use a message-handling system.
- The connections between processing centers are provided by a system of direct links rather than by buses linking many centers together. This implies that each center communicates directly with a relatively small number of other centers (its neighbors). To communicate with a module in a processing center that is not a neighbor, a message will have to be relayed.
- The links between centers are assumed to provide a highly redundant set of paths between any two centers so that the network is not likely to be disjointed by even massive damage.
- It is assumed that the computational resources required by any program module can be supplied by any of several processing centers. Also, the total processing capacity in the system is expected to be substantially more than that required by the application program, providing that severe damage has not occurred or that too many faults have accumulated. It is expected, therefore, that significant freedom exists in the assignment of program modules to processing centers. It is this freedom that is exploited to make the system fault-tolerant and survivable, as well as to achieve other system goals.

As stated earlier, the network executive is the key element in making the concept viable; therefore, it must be protected from failure. The network executive is expected to be distributed over the network, with each processing center executing its own component of the network executive. In addition, it is essential that the network executive exhibit the following characteristics:

Global Stability--The actions of the separate components of the distributed network executive should interact in a way that ensures the global stability of the system.

Locally Controlled--Each component of the network executive should use only information about the states of the processing center executing the component and the neighbors of that center.

Locally Effective--The actions of any component should directly affect only the state of the processing center executing the component and one or more of its neighbors.

The CHAMP architecture, as specializied for avionics, is designed to provide survivable computing in the presence of hostile damage. Because the basic design is modular, CHAMP is well suited to modular tasks such as those that comprise the avionics tasks. Avionics can be thought of as a group of antonomous but related functions tied together by the display and control functions in the cockpit of an aircraft. Some examples of avionic functions are flight control, navigation, radar, communications, ECM, and weapons.

CHAMP is a simple network of autonomous but related processors, called processing centers (PC). A PC is a complete computer containing all elements such as memory, arithmetic logic units, and program control units found in a usual computer. Autonomous means that each processor works independently to make decisions based on its own perception of its working environment. The processors are joined by direct neighbor-to-neighbor communication paths (as many as are needed for the communications and survivability requirements of the application). The size and connectivity of a CHAMP network is one of the design problems presented by any specific application.

In addition to being autonomous, the processors are of such modest capability that within the next five years they may be implemented on a single silicon chip.

The connectivity among the computers is relatively simple; therefore, the chips themselves would need very few pins (connections). This helps to enhance reliability of the overall computer. In most applications, a single CHAMP processor chip would require no more than 20 pins, compared to other multiprocessor or multichip organizations which, in VHSIC implementations, would require 200 or more pins per chip. The more connections that are needed, the more difficult it is to achieve reliability.

Non-homogeneous program modules are mapped onto a random but homogeneous network of processors. By homogeneous we mean that each processor behaves with respect to all other processors identically, not that the interior construction of each processor is the same. In

fact, the interior make-up of each chip may be quite different among processors. For instance, newer chips may be used to replace failed older chips and might be considerably more capable and faster than the older chips, and yet would serve the same function.

Figure 2 shows a simple network of PCs depicted by squares. When necessary, the PCs can be arranged into groups which are called neighborhoods, and which consist of processors with similar specialized characteristics in order to cooperate during related functions. The neighborhoods A, B, and C in Figure 2 are connected by communications lines, but not as heavily as the connection among processors inside the neighborhood.

The arrangement of PCs into neighborhoods corresponds to physical reality in the aircraft environment. One neighborhood might be used to process engine control functions, and therefore be located near the engine pod, while another group might be used to process a radar signal data stream, and therefore be mounted near the radar receiver. These two neighborhoods probably would be located a great distance from one another; therefore, they could not have as many connections between them as the processors could have among themselves within a single neighborhood.

There are several benefits to be gained from having neighborhoods formed to process related functions. First, the communications among PCs is simplified by the short distances involved. Second, restructuring in the case of damage is facilitated. At the same time, the assets of a remote neighborhood may be called into play should they be needed in case of excessive damage to a particular neighborhood. For instance, there might be enough damage in the radar-receiver computer to prevent the radar neighborhood from processing all needed functions. Therefore, the radar altimeter neighborhood might be called on to supply some of its spare resources to assist in the radar signal processing. Signal processing is a very high-speed function, and therefore probably would have to slow down or gracefully degrade if the PCs in the other neighborhoods are needed. These spares might not be as capable as the primary processor of such a high operation rate.

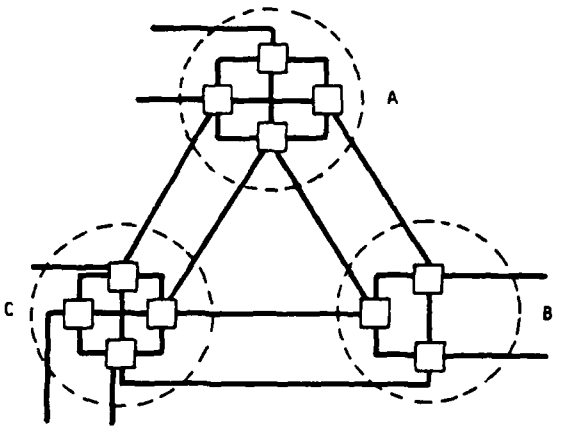


FIGURE 2 NONHOMOGENEOUS PROGRAM MODULES MAPPED ONTO A RANDOM HOMOGENEOUS NETWORK

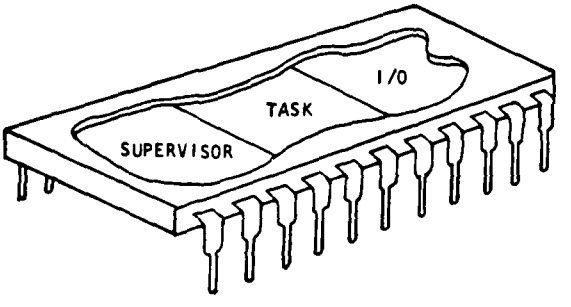


FIGURE 3 DIAGRAM OF A PC SHOWING THREE PROCESSORS

A PC is a complete computer. In addition, it is a computer that contains at least three processors or processes, depending upon the physical design of the chip (Figure 3). The three processes are: the supervisory function, which runs the standard system programs and, in addition, is the basic fault recovery manager for the computer; the input/output (I/O) processor, which connects the PCs via the communications links; the task processor, which accomplishes the user task functions (those functions for which the computer was originally needed).

The operating system is completely distributed in the CHAMP concept. System behavior is duplicated in every PC. The supervisor and the I/O processor cooperate together to form the fault-tolerating operating system, and their behavior is invisible to the applications programmer.

The I/O processor provides communications to neighbors or peripherals, and can be described as the backbone of the network. Figure 4 shows the features of the I/O processor. All the I/O processors together form the basic CHAMP network. The task and the supervisor processors can be thought of as subscribers to this communicating network, similar to subscribers to a telephone network or subscribers to the ARPA net. Peripherals are connected into the network through the I/O processor. Figure 4 shows a radar and tape unit peripherals connected to an I/O processor.

The task processor executes the application program in an entirely ordinary way. Its only extraordinary feature is that the modules of program code are generally smaller in the CHAMP network than most programs written for sequential machines.

The task processor is tailored to perform the application efficiently. This tailoring is an engineering task which must take into account the needs of the application environment. For instance, in Figure 5 the task processor is shown with augmentation to perform functions that would be used in signal processing. Note the Discrete

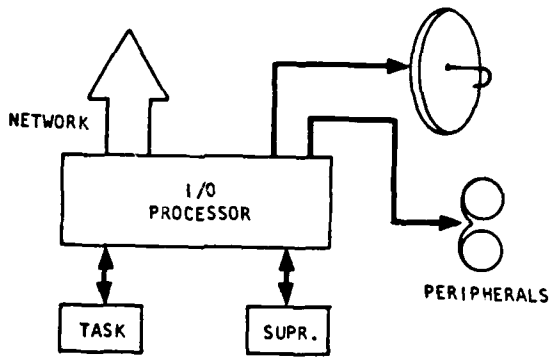


FIGURE 4 DIAGRAM OF HOW THE I/O PROCESSOR PROVIDES COMMUNICATION TO NEIGHBORS AND/OR PERIPHERALS

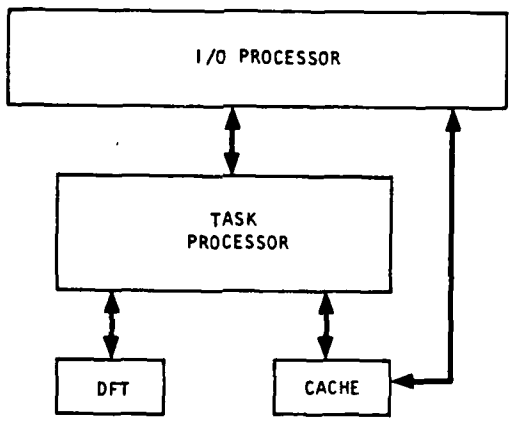


FIGURE 5 DIAGRAM OF HOW THE TASK PROCESSOR CAN BE TAILORED TO PERFORM AN APPLICATION EFFICIENTLY

Fourier Transform (DFT) which is transmitted by satellite to the user processor. Also shown is a cache (or high-speed memory) attached in order to buffer the very high data rates typical of signal processing applications. Because the data rates are so high, the cache memory would probably need to be connected directly to the I/O processor as well as to the task processor. The system is designer is allowed wide architectural freedom in specifying the task processor.

The supervisor processor and, to some degree the I/O processor, make the PCs work together. A very important dimension of this operation is the diagnostics which must be performed to ensure that all elements of the network are functioning well. Each PC is responsible for assessing and maintaining a log of its own condition and the condition of each neighbor directly connected to it.

The supervisor processor periodically schedules diagnostic programs to be run on itself, the local task processor, and the I/O processor. It also schedules diagnostics in each neighbor directly connected to its own PC. Testing does not extend any further than the directly connected neighbor. In this way, each PC has a current record of the health of its own resources and resources to which it is directly connected.

In addition, of course, the supervisor processor performs the usual system functions of load levelling, scheduling, and controlling the other processors. The I/O processor performs a diagnostic test of its neighbors by reporting when a previously good communications link has been found faulty. These algorithms are described in more detail in Section IV, and a design of their detailed behavior is contained in Appendices A and B.

Some avionic computation tasks require signal processing. At the present stage of this system's technology, signal processing occurs at a much higher rate than any single computer could process it. For this kind of application, the current solution is to use many processors arranged in a "pipeline" configuration and provided with special program control elements. The control elements break the incoming

data stream into segments, each of which is given to a separate processor in the pipeline structure. The outputs of all processors then are rejoined together by the control elements to form a continuous data output stream.

The usual signal processor is a hardware construction created specifically to perform this commutation, parallel processing, and the decommutation (reassembly) process. A major problem with such specialized processors is that they are relatively vulnerable to failure because any segment of this complex circuit can fail and render the processor itself useless.

CHAMP, however, is well suited to this type of processing because there are many spare resources available, and the failure of any one of the pipeline processors does not damage the overall performance. In the CHAMP structure, the incoming data stream arrives at one or two (or more) PCs and is commutated into manageable packets in the same way the specialized pipeline processor breaks up the incoming data stream. These data packets then are distributed throughout the network to PCs which perform specialized signal processing operations such as Fourier transforms, inverse Fourier transforms, and Hilbert transforms, etc. When these operations are completed at the various sites, a decommutation process--which is a user process located in the network--gathers the several output data streams and forms them into a continuous output stream exactly as the pipeline signal processor would have done. Figures 6 and 7 show a schematic comparison between one type of a pipeline processor and the CHAMP structure.

A major design problem in the signal processing is the very high data rate at the incoming side--somewhere between 1 gigabit and a 100 gigabits per second. The technology for transmission of this kind of a serial data stream presently is in development. This same technology is applicable to the communication processor of the CHAMP PC.

By incorporating the needed processing augmentation and communications speed enhancement, a CHAMP-style architecture is feasible for a pipeline computation process. Once the incoming data stream has been

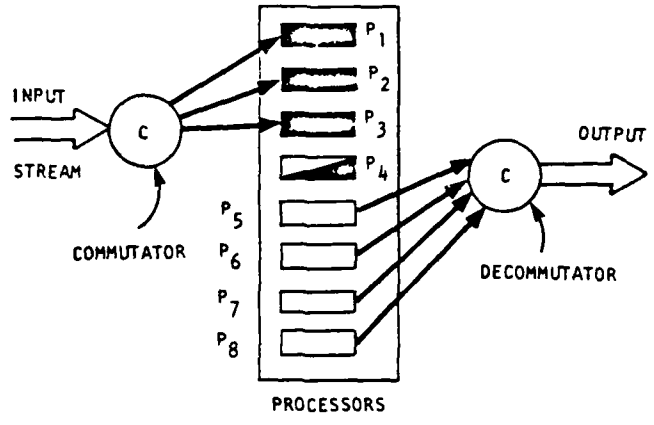


FIGURE 6 DIAGRAM OF TRADITIONAL SIGNAL PROCESSING CONFIGURATION

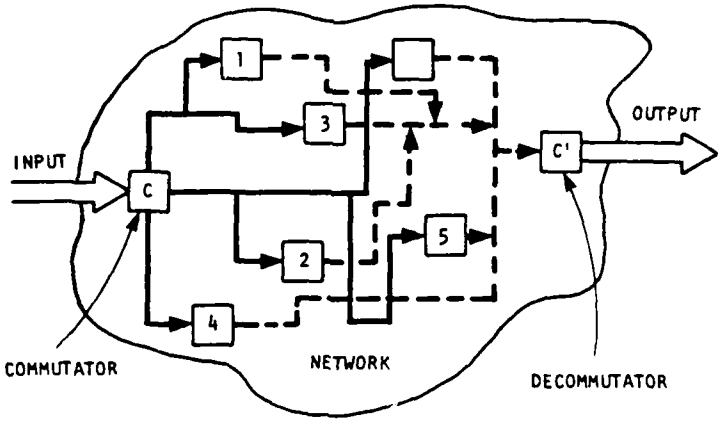


FIGURE 7 DIAGRAM OF PIPELINE PROCESSING IN A CHAMP NETWORK

divided into smaller chunks, the movement of these packets through the network can proceed at a much lower data rate therefore, lower speed links can be used to increase the noise margin (reduce the noise vulnerability). The only portions of the network that must handle a very high data rate are at the input or output.

A distinct advantage to using the CHAMP-sytle generalized network for signal processing is that the programming of such a network to perform a variety of signal processing problems is greatly simplified over what would be required in a special-purpose pipeline processor. The programmer in the CHAMP system need not know how the data stream is handled at the bit level. Furthermore, the CHAMP hardware is not peculiar to a class of applications. Should the need arise to make a slight change in the application processing, it can be accomplished with a direct software change in most cases. Therefore, the expense and time required to make the hardware modification on a pipeline processor can be avoided.

In Figure 6, the input data stream is shown arriving at a traditional pipeline processor commutator, where it is broken into segments which are distributed sequentially (processor 1, processor 2, processor 3 etc.) one after the other. The decommutation must take place after the processors are finished; Figure 6 shows the collection of completed data output from processors 5, 6, 7, and 8, to form the output stream. Processor 4 is already empty and awaiting new data. In Figure 7, the equivalent situation is shown in a CHAMP network, with the input data arriving at the commutation function housed in one PC. These data are then broken into packets and shipped sequentially to PCs marked 1, 2, 3, 4, etc., in the diagram. Decommutation is performed at another PC programmed to gather the output from each of the PCs as it is available and to form it into a continuous output stream.

III AVIONICS SOFTWARE

Software developed using modern structured design techniques yields modules of code that have well defined inputs and outputs. The resulting modules are small enough to be distributed in a multiprocessor network such as CHAMP with I/O operations being handled through communications between processing centers.

The avionics software used in this study had been previously modularized by the Air Force DAIS project using structured techniques. The resulting code presented each function as a number of small and manageable modules. For example, Figure 8 shows the major modules of the rudder command: rudder control, lateral accelerator, lateral integrator, YAW trim and YAW rate. Each of these modules of code is small compared to the capability of a modern microprocessor; therefore, several of these functions can be run in a single processor and the time requirements still can be met. Since each of these modules of code is so small, and since the interaction among them is limited to two or three other modules, it is reasonably easy to distribute these modules of code in a connected and communicating network of processors to allow each module to perform its job in a timely fashion, to allow the entire rudder command section to share the resources of many computers.

In the performance of this contract, two avionic functions (navigation and flight control) were divided into such modules, called relocatable units (RUs) of code. The information used in this portion of the study was supplied by AFWAL; navigation function data are taken from DAIS documents and listings, and flight-control function data were taken from similar documents and listings of the DAIS flight control simulation algorithm from the Flight Dynamic Laboratories.

Navigation functions process data from five sources: the radar altimeter, the ILS, TACAN, INS and air data sensors. These five inputs are processed to create an updated navigation state. Figure 9 shows the interrelationship of these code modules.

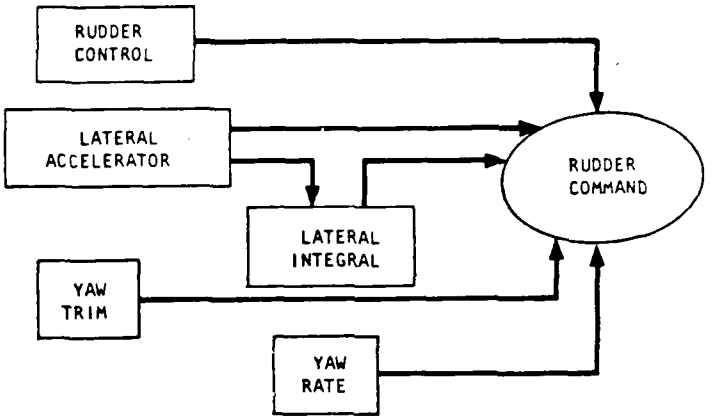


FIGURE 8 DIAGRAM OF MAJOR AVIONICS SOFTWARE MODULES TO CONTROL RUDDER

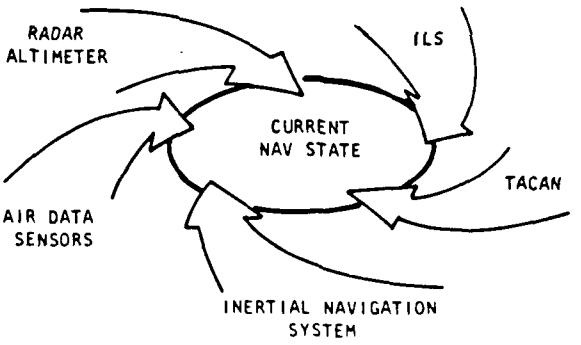


FIGURE 9 DIAGRAM OF DATA SOURCES FOR NAVIGATION FUNCTIONS

Flight control is managed as three separate functions: pitch, yaw, and roll. Figure 10 describes the interrelationship of these three functions.

Tables 1 and 2 show the message data-block interchange among the navigation processing tasks. All data blocks given in the source document, and which are used exclusively by the task that creates the data, were eliminated from the interchange diagram. For cases where other tasks use such internal data, a "write" (W) is indicated in the table where "update" (U) is used by the source documents.

Tables 3 and 4 list the specifications for each RU of the DAIS navigation function. Some simplification of the communication specification was made for this study. Note that SP11 outputs an event (E449) that initiates SP12. SP12 also can be initiated by event E450. Several events are outputs of SP12; for the purpose of this study they are all represented by event E451. External data indicated in the tables are treated as though they were data messages coming from or going to RUs of other avionics functions, such as stores management.

The execution times and period/phase values of Tables 3 and 4 relate to the specific implementation on the DAIS project computer. Period values refer to the number of minor cycles per second, and phase designates where in the major cycle the execution begins (one minor cycle is 8 ms in length, and 32 minor cycles comprise a major cycle of 256 ms). If we assume processor characteristics for the CHAMP lattice PC, we can normalize the DAIS numbers to fit the CHAMP PC characteristics. As a starting point, we have assumed that the PCs in the CHAMP lattice will consist of 16-bit microprocessors with 16K words of memory available to process RUs. Each PC is assigned an average instruction execution speed of one half that of the DAIS computer (e.g., SP05 takes 4.8 ms in the DAIS computer and would require 9.6 ms in a CHAMP PC). Since task processing will be performed in parallel as opposed to time-sharing as in the DAIS computer, execution time becomes less critical.

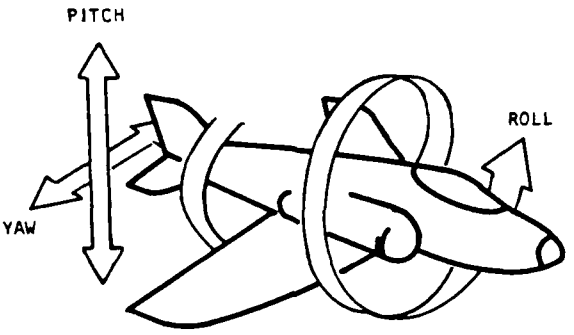


FIGURE 10 DIAGRAM OF THREE FUNCTIONS AFFECTING FLIGHT CONTROL

Table 1

DATA BLOCK INTERCHANGE OF PROCESSING TASKS (RUs)
FOR DAIS NAVIGATION FUNCTION

NAVDAT Compool (NC)

Data Block No.	01	02	06	07	08	09	11	12	13	15	16	18	20	26	27	28	29	30	32	33	37	40	41	51	52	55	74	75	76	77	78
No. of Words	2	10	4	12	14	8	6	8	6	12	8	8	10	6	10	9	12	4	14	11	4	4	10	8	4	19	6	4	16	10	12
Processing Tasks (RUs)																															
SPØ2		R			R			R												R		W									
SPØ3				R		R																							W		
SPØ4			R						R						R				R											W	
SPØ5			R		R					R	R		R							R											W
SPØ7	R				W			R											W	W	W		R						R	R	R
SPØ8					R									R																	
SP11					R				R	R	R			R	R	R	W	W				R		R	R	R/W	R	R			
SP12					R									W											R	R/W	W				
SP20			W							R	W	W																			
SP21					R					R	R		W							R											
QPØ2			R																												
QPØ3				R/W			R																								
QPØ4		W		W		W		W	W														W								
QP15																															
QP16															W																
QP20																															
QP21																W															
QP25																															
QP26																				W											
QP40										W																					

R - Read
W - Write

Table 2

DATA BLOCK INTERCHANGE OF PROCESSING TASKS (RUs)
FOR DAIS NAVIGATION FUNCTION (SHEET 2)

	QPMODE COMPOOL (QC)					Health COMPOOL (HC)	
Data Block No.	02	04	05	08	10	04	40
No. of Words	4	20	7	7	8	5	2
Processing Tasks (RUs)							
SPØ2							
SPØ3				R			
SPØ4	R						
SPØ5							
SPØ7							
SPØ8	R/W						
SP11		R					
SP12							
SP20			R				
SP21							
QPØ2							
QPØ3							
QPØ4				W			
QP15	R						
QP16	W						
QP20		R				W	
QP21							
QP25	R						
QP26							W
QP40				R			

R-READ
W-WRITE

Table 3

SPECIFICATION OF RELOCATABLE UNITS FOR NAVIGATION FUNCTION PROCESSING

Relocatable Units (RUs)		Input Data from		Output Data to		DAIS	Execution Time	
DAIS Notation	RU Description	RUs	No. of Words	RUs	No. of Words	Priority	(ms)	Period/Phase
SP02	Computes navigation parameters	QP04	18	SP11	10	-9	3.0	32/16
		SP07	25					
SP03	Computes position and velocity from INS data	QP03	12	SP07	16	-2	3.0	4/2
		QP04	28					
SP04	Computes lat., long., & hor. vel. from range and bearing of TACAN	SP07	14	SP07	10	-3	4.0	16/9
		SP08	20					
		SP20	4					
		QP04	6					
		QP16	30					
SP05	Computes position and vel. by dead reckoning navigation	SP07	25	SP07	12	-5	4.8	32/7
		SP20	12					
		SP21	10					
		QP40	12					
SP07	Computes best available navigation source	SP03	16	SP02, SP05,				
		SP04	10	SP21	25	-6	1.0	4/0
		SP05	12	SP04	14			
		QP04	18	SP08, SP11, SP12	14			
		External*	2	External*	4			
SP08	Computes TACAN station for TACAN navigation	SP07	14	SP04, QP15	20	-8	2.0	128/1
		SP12	6					
		QP16	20					
SP11	Computes flight path deviations	SP02	10	SP12	19	-1	0.7	16/9
		SP07	14	External*	16			
		SP12	30	Event 449				
		QP04	14					
		QP16	10					
		QP21	9					
		QP40	12					
		External*	23					
SP12	Processes flight plan data and pilot input mode requests	SP07	14	SP08	6	-2	2.0	Async. initiated by event 449 or 450
		SP11	14	SP11	30			
		External*	4	Event 451				
		Event 449 or 450						
SP20	Computes flight parameters from data supplied by air data sensors	QP40	12	SP05	12	-4	3.4	16/11
		External*	7	SP04, QP02	4			
				SP21	8			
				External*	8			
SP21	Computes wind vector using air data, inertial velocities and airframe transformation matrix	SP07	25	SP05	10	-7	8.4	16/15
		SP20	8					
		QP40	12					

*These data are input/output from/to external processes

Table 4

SPECIFICATION OF RELOCATABLE UNITS FOR NAVIGATION FUNCTION PROCESSING

Relocatable Units (RUs)		Input Data from		Output Data to				
DAIS Notation	RU Description	RUs/Equip.	No. of Words	RUs/Equip.	No. of Words	DAIS Priority	Execution Time (ms)	Period/Phase
QP02	Writes barometric altitude to INS	SP20	4	INs	11	P	0.1	16/7
QP03	Writes dir. cos. matric and position updata to INS	QP04	12	INS	17	-15	2.2	Async. data initiated
		External*	6	SP03	12			
QP04	Reads INS data	INS	29	SP02	18	P	3.0	4/2
				SP03	28			
				SP04	6			
				SP07	18			
				SP11	14			
				QP03	12			
QP15	Writes current TACAN state	QP16, SP08	20	TACAN	2	-38	.86	Async. data initiated
QP16	Reads TACAN data and health status	TACAN	9	SP04	30	-37	1.45	16/9
				SP11	10			
				SP08, QP15	20			
				External*	5			
QP20	Writes current ILS state	External*	7	ILS	3	-39	0.40	Async. data initiated
QP21	Reads ILS data	ILS	6	SP11	9	P	0.20	16/13
QP25	Writes current radar altimeter state	External*	4	Radar altimeter	3	-45	0.48	Async. data initiated
QP26	Reads radar altimeter data and health status	Radar altimeter	3	External*	6	P	0.20	16/1
QP40	Reads data from air data sensors	Air data sensors	5	SP05, SP11, SP20				
		External*		SP21	12	P	0.22	16/11

*These data are input/output from/to external processes

The synchronous RUs are processed periodically at the appropriate minor cycle interval (as shown in Tables 3 and 4), but can start any time that the necessary input data are within the major cycle.

Control features in the CHAMP executive help to ensure that each calculation is completed within its specified period and that data messages are supplied to the RUs requiring the output.

Memory storage requirements for each RU were estimated from assembly program listings of the navigation function modules for DAIS. Table 5 describes the memory storage requirements, execution time and period of the navigation RUs for processing in a CHAMP lattice. The execution time assumes the factor of two relative speed disadvantages over the DAIS computer already mentioned.

Figure 11 shows message flow among RUs for the navigation function. The information in Table 5 and the RU input/output data of Tables 3 and 4 provide the necessary data for mapping the navigation tasks onto a lattice of PCs. The navigation function is a fairly small set of RUs which can easily be mapped onto approximately six or seven PCs. Allowing for 100% backup, only 12 to 14 PCs are needed.

To provide a varied application environment for the CHAMP simulation, the flight control function was added to the processing tasks. Figure 12 shows the data communications for the flight control function. The processing tasks were broken into RUs in the same way as were the navigation functions.

The specifications for the navigation and flight control RUs are given in Tables 6 through 9. These tables show the data inputs, data outputs, survival priority, memory requirements, execution time, processing period, and load for each RU.

Survival priorities for the navigation RUs were provided by the DAIS program office. The values have been recomputed so that the highest priority is indicated by the largest value. Survival priorities for the flight control RUs were determined by SRI and judged by Air Force personnel to be adequate for this simulation.

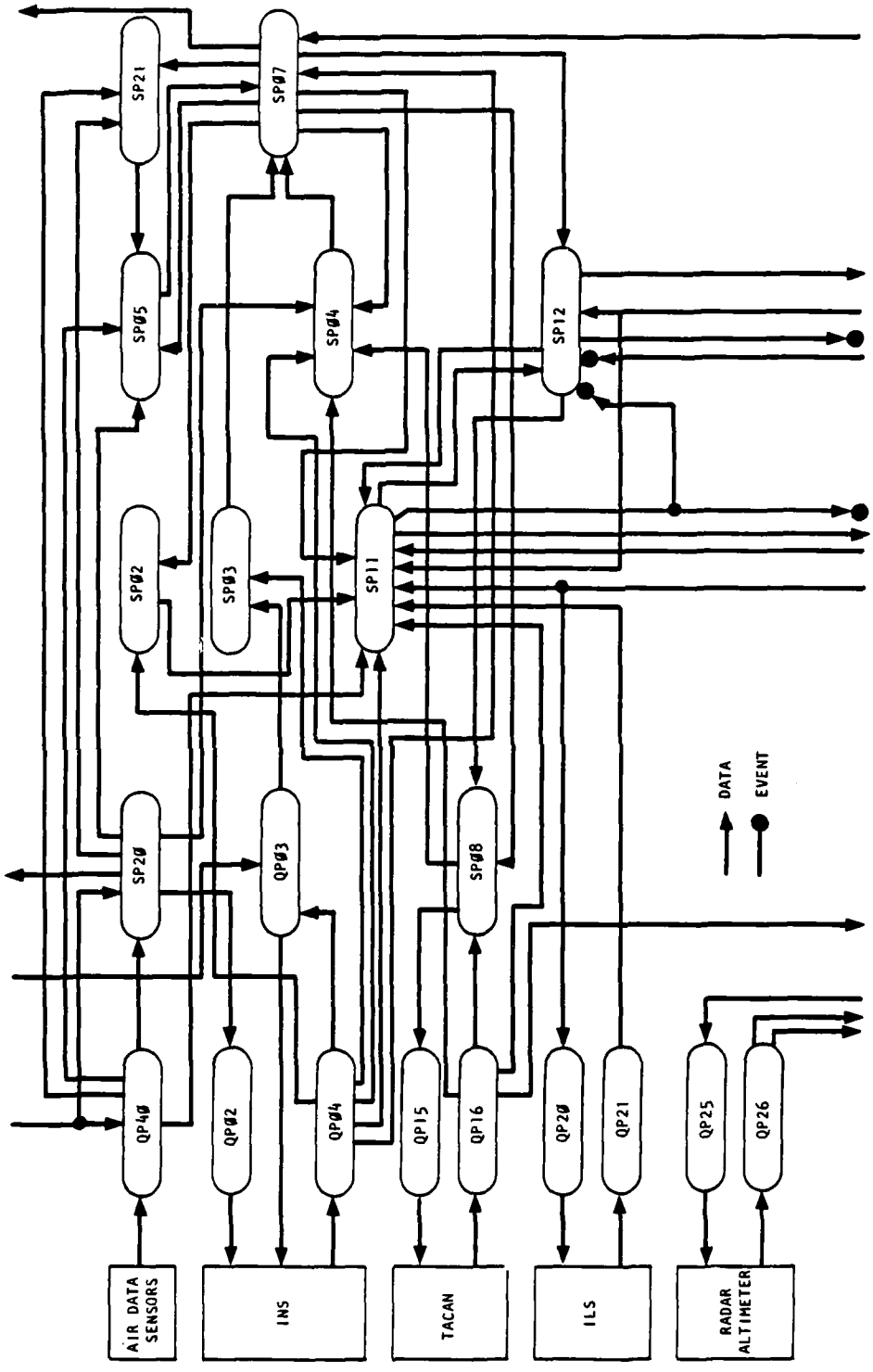


FIGURE 11 DATA COMMUNICATIONS AMONG NAVIGATION RUs

Table 5

DESCRIPTION OF NAVIGATION RUs
FOR PROCESSING IN CHAMP LATTICS

RUs	Memory Storate Requirements (K words)	CHAMP Processor Execution Time (ms)	Period (ms)
SP02	1.4	6.0	256
SP03	1.4	6.0	32
SP04	3.0	8.0	128
SP05	1.6	9.6	256
SP07	1.6	2.0	32
SP08	1.4	4.0	1024
SP11	3.0	1.4	128
SP12	1.5	4.0	Async.
SP20	1.2	6.8	128
SP21	1.2	16.8	128
QP02	0.4	0.2	128
QP03	1.3	4.4	Async.
QP04	1.0	6.0	32
QP15	1.8	1.7	Async.
QP16	1.5	2.9	128
QP20	1.0	0.8	Async.
QP21	0.5	0.4	128
QP25	1.0	0.96	Async.
QP26	0.4	0.4	128
QP40	0.5	0.44	128

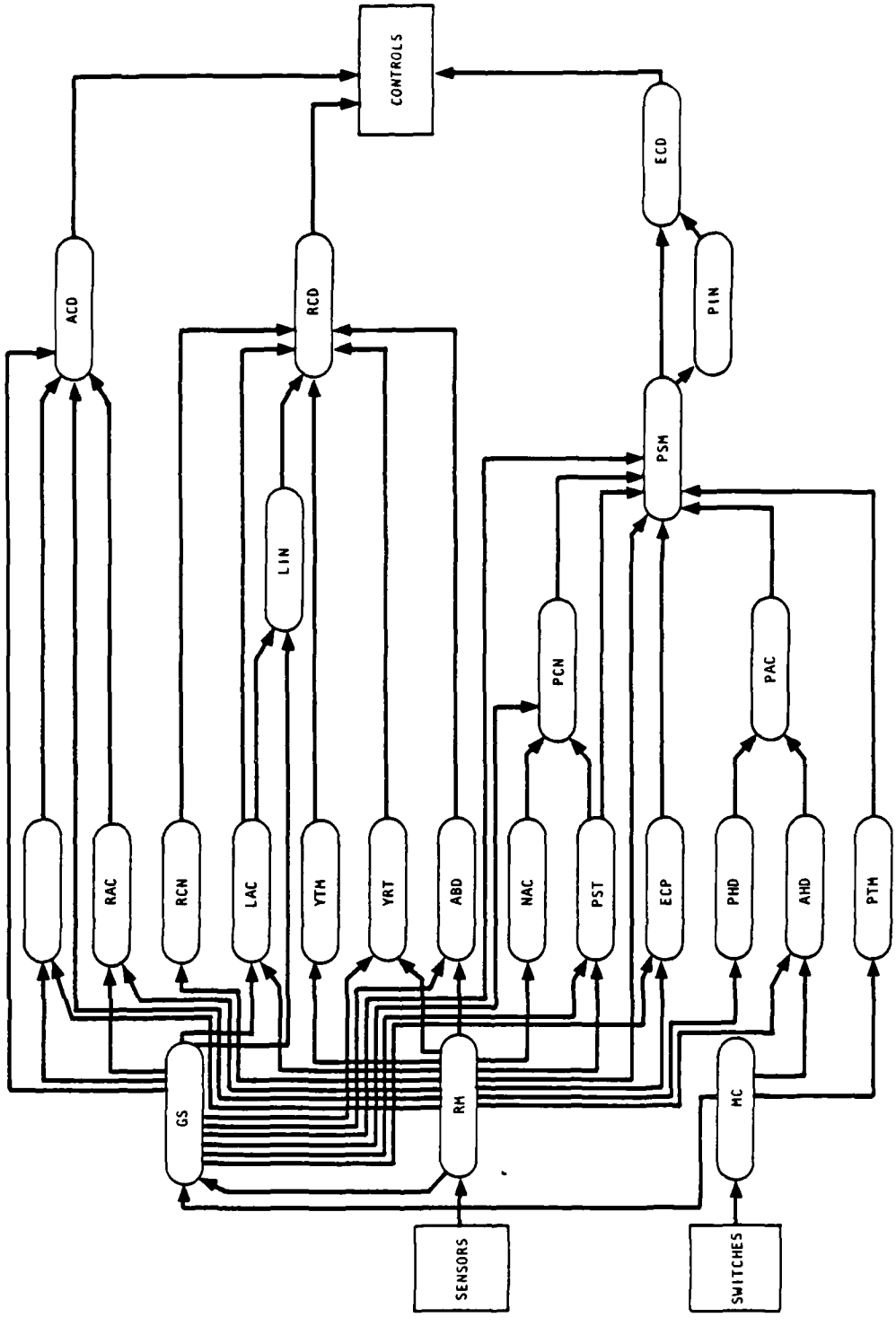


FIGURE 12 DATA COMMUNICATIONS AMONG FLIGHT CONTROL RUs

Table 6

SPECIFICATION OF RELOCATABLE UNITS FOR NAVIGATION FUNCTION PROCESSING

Relocatable Units (RUs)		Input Data from		Output Data to			Memory Storage Requirement (Kwords)	CHAMP Processor Execution Time (ms)	Period (ms)	Load (%)
DAIS Notation	RU Description	RUs	No. of Words	RUs	No. of Words	Survival Priority				
SP02	Computes navigation parameters	QP04	18	SP11	10	70	1.4	6.0	256	2
		SP07								
SP03	Computes position and velocity from INS data	QP03	12	SP07	16	90	1.4	6.0	32	19
		QP04	28							
SP04	Computes lat., long., & hor. vel. from range and bearing of TACAN	SP07	14	SP07	10	80	3.0	8.0	128	6
		SP08	20							
		SP20	4							
		QP04	6							
		QP16	30							
SP05	Computes position and vel. by dead reckoning navigation	SP07	25	SP07	12	81	1.6	9.6	256	4
		SP20	12							
		SP21	10							
		QP40	12							
SP07	Computes test available navigation source	SP03	16	SP02, SP05,						
		SP04	10	SP21	25	88	1.6	2.0	32	6
		SP05	12	SP04	14					
		QP04	18	SP08, SP11, SP12	14					
		External*	2	External*	4					
SP08	Computes TACAN station for TACAN navigation	SP07	14	SP04, QP15	20	64	1.4	4.0	1024	1
		SP12	6							
		QP16	20							
SP11	Computes flight paths deviations	SP02	10	SP12	19	72	3.0	1.4	120	1
		SP07	14	External*	16					
		SP12	30	Event 449						
		QP04	14							
		SP16	10							
		QP21	9							
		QP40	12							
		External*	23							
SP12	Processes flight plan data and pilot input mode requests	SP07	14	SP08	6	68	1.5	4.0	Async	1
		SP11	14	SP11	30					
		External*	4	Event 451						
		Event 449 or 450								
SP20	Computes flight parameters from data supplied by air data sensors	SP40	12	SP05	12	92	1.2	6.8	128	5
		External*	7	SP04, QP02	4					
				SP21	8					
				External*	8					
SP21	Computes wind vector using air data, inertial velocities and airframe transformation matrix	SP07	25	SP05	10	84	1.2	16.8	128	13
		SP20	8							
		SP40	12							

*These data are input/output from/to external processes

Table 7

SPECIFICATION OF RELOCATABLE UNITS FOR NAVIGATION FUNCTION PROCESSING (Concluded)

Relocatable Units (RUs)		Input Data from		Output Data to		Survival Priority	Memory Storage Requirement (Kwords)	CHAMP Processor Execution Time (ms)	Period (ms)	Load (%)
DAIS Notation	RU Description	RUs/Equip.	No. of Words	RUs/Equip.	No. of Word					
QP02	Writes barometric altitude to INS	QP20	4	INS	11	92	0.4	0.2	128	1
QP03	Writes dir. cos. matrix and position update to INS	QP04 External*	12 6	INS SP03	17 12	76	1.3	4.4	Async	1
QP04	Reads INS data	INS	28	SP02 SP03 SP04 SP07 SP11 QP03	18 28 6 18 14 12	96	1.0	6.0	32	19
QP15	Writes current TACAN state	QP16, SP08	20	TACAN	2	80	1.8	1.72	Async	1
QP16	Reads TACAN data and health status	TACAN	9	SP04 SP11 SP08, QP15 External*	30 10 20 5	80	1.5	2.9	128	2
QP20	Writes current ILS state	External*	7	ILS	3	72	1.0	0.8	Async	1
QP21	Reads ILS data	ILS	6	SP11	9	72	0.5	0.4	128	1
QP25	Writes current radar altimeter state	External*	4	Radar altimater	3	56	1.0	0.96	Async	1
QP26	Reads radar altimeter data and health status	Radar altimeter	3	External*	6	56	0.4	0.4	128	1
QP40	Reads data from air data sensors	Air data sensors External*	5 12	SP05, SP11, SP20 SP21	12	92	0.5	0.44	128	1

*These data are input/output from/to external processes

Table 8

SPECIFICATION OF RELOCATABLE UNITS FOR FLIGHT CONTROL FUNCTION PROCESSING

Relocatable Units (RUs)	Input Data from		Output Data to		Survival Priority	Memory Storage Requirement (Kwords)	CHAMP Processor Execution Time (ms)	Period (ms)	Load (%)
	RUs/Equip.	No. of Words	RUs/Equip.	No. of Words					
Redundancy management (RM)	Sensors	33	RET	1	96	1.0	2.0	12.5	16
			ACD	1					
			RAC	2					
			GS	1					
			RCN	1					
			LAC	1					
			LAC, YTM	1					
			YRT	1					
			YRT, ABD	1					
			ABD, PSM	1					
			ABD, NAC	1					
			ABD	1					
			ABD, GS	1					
			ABD, ECP, PHD, RAC	1					
			ABD, ECP, PHD	1					
			PST	1					
			AHD	1					
Mode control (MC)	Switches	4	PTM	1	1	0.5	0.8	100	1
			GS	3					
			AHD						
Altitude hold (AHD)	RM	1	TAC	1	4	0.4	0.8	50	2
	MC	1							
Pitch A/P command (PAC)	PHD	1	PSM	1	52	0.2	0.5	25	2
	AHD	1							
Up elevator compensation (ECP)	RM	2	PSM	1	48	0.7	1.8	25	7
	GS	2							
Roll A/P command (RAC)	RM	3	ACD	1	44	0.8	0.2	25	1
	GS	2							
Gain schedule (GS)	RM	2	PCN	1	92	0.4	1.1	100	1
	MC	3	ECP, PST	2					
			PSM	2					
			LAC, LIN, ABD, ACD	1					
			RAC	2					
			RST	1					
			YRT	1					
Pitch altitude hold (PHD)	RM	2	PAC	1	8	0.2	0.6	25	2

Table 9

SPECIFICATION OF RELOCATABLE UNITS FOR FLIGHT CONTROL FUNCTION PROCESSING (Concluded)

Relocatable Units (RUs)	Input Data from RUs/Equip.	Input Data from No. of Words	Output Data to RUs/Equip.	Output Data to No. of Words	Survival Priority	Memory Storage Requirement (Kwords)	CHAMP Processor Execution Time (ms)	Period (ms)	Load (%)
Pitch trim (PTM)	MC	1	PSM	1	16	0.3	0.6	50	1
Normal acceleration (NAC)	RM	1	PCN	1	88	0.4	1.0	25	4
Pitch stick (PST)	GS	2	PCN	1	40	0.5	1.2	25	5
	RM	1	PSM	1					
Pitch control (PCN)	GS	1	PSM	1	36	0.3	0.7	25	3
	NAC	1							
	PST	1							
Pitch sum (PSM)	GS	2	ECD	1	32	0.4	1.0	12.5	8
	PTM	1	PIN						
	PCN	1							
	PST	1							
	RM	1							
	ECP	1							
	PAC	1							
Pitch integral (PIN)	PSM	1	ECD	1	20	0.3	0.7	12.5	6
Elevator command (ECD)	PSM	1	Control	1	64	0.2	0.4	12.5	3
	PIN	1							
Roll stick (RST)	GS	1	ACD	1	28	0.5	1.2	25	5
	RM	1							
Aileron command (ACD)	GS	1	Controls	1	60	0.3	0.6	25	2
	RM	1							
	RST	1							
	RAC	1							
Yaw trim (YTM)	RM	1	RCD	1	12	0.2	0.3	25	1
Lateral acceleration (LAC)	GS	1	LIN	1	80	0.4	1.0	25	4
	RM	2	RCD	1					
Lateral integral (LIN)	GS	1	RCD	1	76	0.5	1.0	25	4
	LAC	1							
Rudder control (RCN)	RM	1	RCD	1	24	0.3	0.6	25	2
Yaw rate (YRT)	RM	2	RCD	1	72	0.2	0.5	25	2
	GS	1							
Alpha blend (ABD)	GS	1	RCD	1	68	1.4	3.8	50	8
	RM	1							
Rudder command (RCD)	RCN	1	Controls	1	56	0.3	0.7	25	3
	LAC	1							
	LIN	1							
	YTM	1							
	YRT	1							
	ABD	1							

Memory requirements, which include space both for storage and for processing, were taken from the original program listings and include memory needed for all subroutines called by each program module. Execution times were estimated by a manual count of the instruction execution times for each RU (including all subroutines called by that RU). These times then were normalized to fit the CHAMP PC processing characteristics. For this purpose, the instruction times given in the DAIS Processor Instruction Set document were used.

The load value is the percentage of processor capacity that a particular RU requires to complete its processing within the specified time period betweem executions. This percentage is obtained by dividing execution time by the period. Therefore, maximum load a PC can process is 100. The load value is used by the load-levelling algorithm of the operating system to maintain all PCs in the network at an equivalent load.

In sizing the task processor of the PC to contain 16K words of memory, diagnostics are given approximately the same emphasis as application task processing. Although diagnostic requirements are somewhat application dependent, there are two factors that make this emphasis important. First, it is important to the continuous functioning of a fault-toleranting system that only functional resources be invoked when failures occur. If in the process of recovery from a failure a faulty processor is invoked, there is no guarantee that recovery will be effective. Second, the cost of hardware is small compared to overall system costs or costs incurred as a result of failures. Higher reliability through more extensive diagnostics is achieved by adding more hardware (spare resources) to the network at a small cost, to provide more time for diagnostics.

There is presently no measure for determining the amount of diagnostics that must be performed to achieve a given degree of reliability. The amount is chosen as a tradeoff between reliability requirements and system cost. For the avionics case, the lattice was

sized to allow half the processing time and space to diagnostics. This decision was arbitarily made based on the importance of flight control. It is assumed that not all PCs are actively processing. At least in the absence of failures, spare PCs are being exercised wide diagnostics.

The memory requirements for the navigation and flight control functions are 27K words and 11K words, respectively. The simulation was designed for a lattice size that provides enough resources to have three backup copies for each RU; however, the actual number of backup copies for given a RU is assigned according to the priority of that RU. Four times the number of navigation RUs requires a total of 108K words of memory, and four times the number of flight control RUs requires a total of 44K words of memory. Dividing these memory values by 8K (the memory initially allocated for RU processing in the task processor of each PC) we find that 14 PCs are required for navigation and 6 for flight control--a total of 20 PCs. Table 10 shows the memory and PC requirements for the navigation and flight control functions. These estimates were used as the basis for the simulation studies that follow.

The navigation and flight control modules share the spare resources in the CHAMP network. For instance, in an actual application, the navigation neighborhood (group of processors) is constructed with a certain spare capability determined by the design survival level. The flight control processor group would have a corresponding set of spare units. This situation is shown in Figure 13. Should damage or natural failures occur to the extent that the spares allocated to any one function are completely used, the spares of the other function can be called into service through the operating system of the CHAMP network.

Based on the results of the sizing study shown in the previous tables, a modest microprocessor based PC could serve as a building block for a CHAMP-style avionics network. As previously mentioned,

Table 10

MEMORY AND PC REQUIREMENTS FOR NAVIGATION
AND FLIGHT CONTROL FUNCTIONS

Avionics Functions	Memory Req. (K Words)	X4 for Spares	X2 for Diagnostics	# PCs
NAV	27	108	216	14
Flight Control	11	44	88	6
	Total PCs in Network = 20			

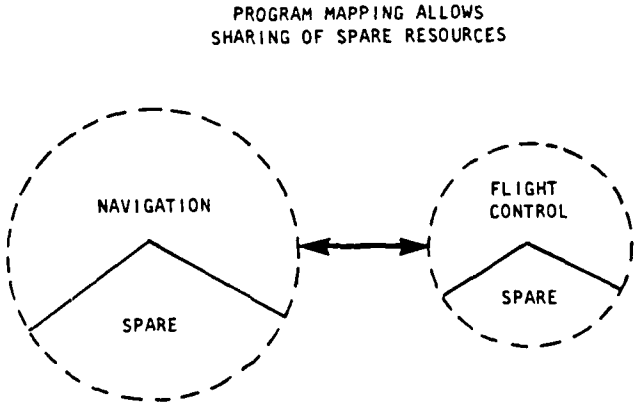


FIGURE 13 DIAGRAM OF SHARED SPARE RESOURCES FOR NAVIGATION AND FLIGHT CONTROL MODULES IN THE CHAMP NETWORK

preliminary information indicates that the task processor would not need more than 16K words of 16-bit memory, the supervisor processor would need about 8K words, and the I/O also about 8K words. Figure 14 shows four I/O ports; however, our studies indicate that five or perhaps six ports might be desirable for adequate survival at a 50% damage level.

Figure 15 shows the basic network that was used in the simulation studies. Results of the simulation runs reported in Section V show that graceful degradation can be obtained if the network remains connected after being damaged. As can be seen in Figure 15, it would be relatively easy to divide or fragment a network connected in this way. The addition of two I/O ports, (for a total of six) would make a much cleaner connection pattern and therefore would make separation of the network into fragments more difficult. Figure 16 shows one possible alternative network constructed from six-port centers.

Finally, it is useful to point out that the CHAMP network is entirely compatible with the existing MIL STD 1553 dual aircraft bus. Figure 17 diagrams how the network might be connected to these two buses through remote terminals. Although two separate terminals (each connected to a separate function) are shown in the figure, the communications routes among the PCs allow either remote terminal to be used for either function. This provides an additional degree of fault toleration.

It needs to be pointed out, however, that the best survival situation would be obtained by having more than two independent buses connecting islands of computing power, (i.e., better survival is obtained if the 1553 is not used within the aircraft computer network). The CHAMP network philosophy is that buses should join only two computers. More than two processors on a single bus increases the possibility of bus failure since the failure of only one processor could incapacitate the bus. Modern communications techniques are reaching a point of simplicity that would allow multiple redundancy in the buses throughout the aircraft with minimum cost, complexity, and weight.

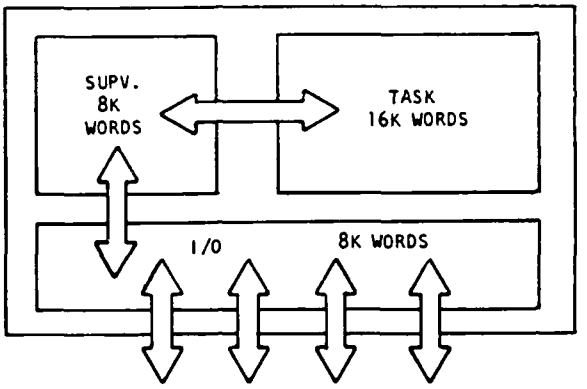
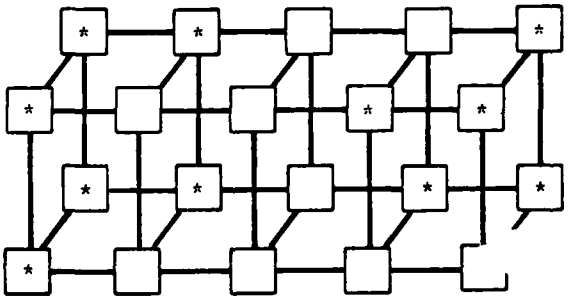


FIGURE 14 DIAGRAM OF MICROPROCESSOR-BASED PC WITH FOUR I/O PORTS



*FAILED PCs

FIGURE 15 DIAGRAM OF BASIC PROCESSING NETWORK USED IN SIMULATION STUDIES (NAVIGATION AND FLIGHT CONTROL)

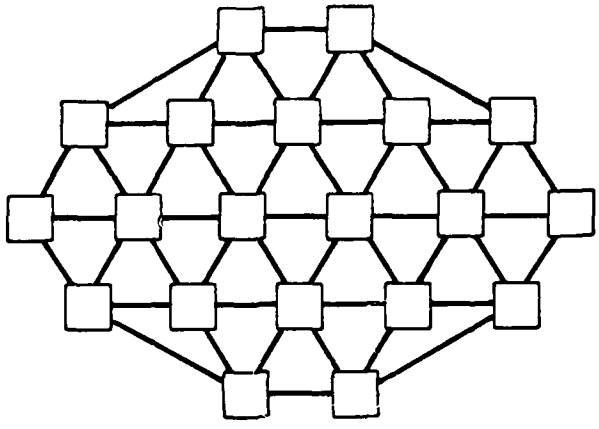


FIGURE 16 A POSSIBLE ALTERNATIVE NETWORK CONSTRUCTED FROM SIX-PORT CENTERS

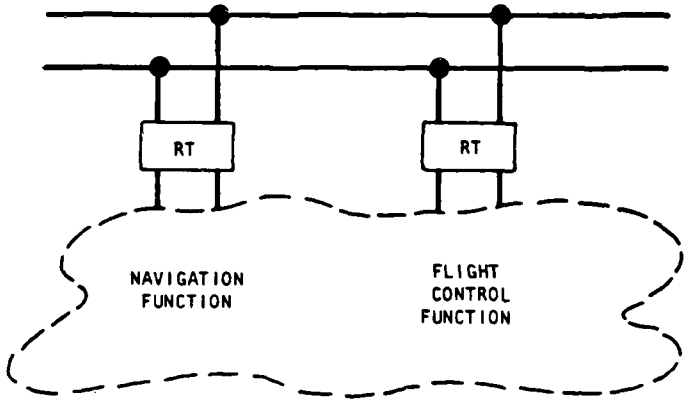


FIGURE 17 SCHEMATIC DIAGRAM OF HOW THE NETWORK MIGHT BE CONNECTED TO THE TWO 1553 BUSES THROUGH REMOTE TERMINALS

IV COMMUNICATIONS ALGORITHMS

This section continues a brief overall description of the communication system in the CHAMP network. More detailed information is provided in Appendix A, in which the structured program design for the control algorithms is described, and in Appendix C entitled "Control of Communication in a MUH Processor Network".

Communications algorithms are designed to manage message traffic among RUs in the network so the applications program designer need not be concerned with their behavior. For instance, Figure 18 shows two specific RUs communicating through three intermediate PCs. In this case, the alpha blend RU has sent a message to the rudder command RU. The route is specific and the message is directed from PCs to PC by the operating system. The pattern of events during processing startup, the sequence of failures, and the detailed connectivity structure at times may make the routes somewhat circuitous, but they will tend to be mimimum in length. Rerouting in case of failure is managed by the I/O processor in the manner described below.

Messages are communicated in one of four modes in the CHAMP lattice: search, directed, default, and broadcast. The mode for an RU to communicate with another RU whose location is unknown is the search mode. At startup time, no PC has route information on how to direct a message to an RU located in another PC. The operating system places this knowledge in each I/O processor as it is found.

1. Search Mode

In the search mode, a message is sent to every PC in the system looking for the target RU. This message eventually will reach the targeted RU. When it does, an acknowledgement message will be returned to the sender. At each point along the path, the intervening PCs have noted the location from which the original message came; therefore, the acknowledgement returns along the most direct route to the sender.

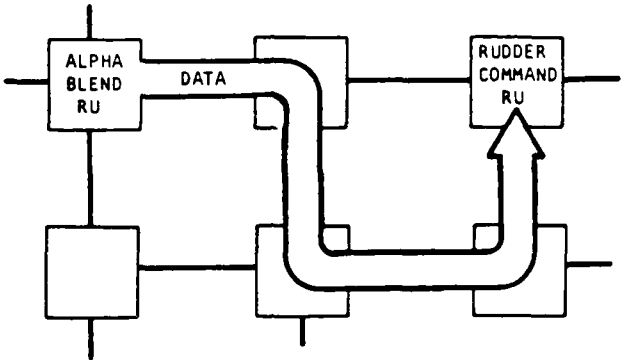


FIGURE 18 DIAGRAM OF TWO SPECIFIC RUs COMMUNICATING THROUGH THREE INTERMEDIATE PCs

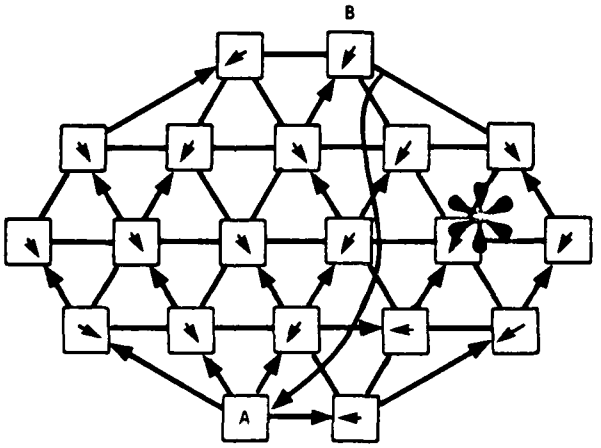


FIGURE 19 DIAGRAM OF SIMPLE NETWORK OF PCs SHOWING ROUTING OF A SEARCH MODE MESSAGE BETWEEN RUs

A search message dies when all PCs have seen it at least once, (each PC will refuse to receive and act upon the message more than once). The first time a PC receives the search mode message, it sends the message out (rebroadcasts it) to all I/O ports except the one from which it was received. At the same time, the I/O processor makes a note of the port from which the message first arrived; this port is the direction toward which a message might be sent to the originating RU.

Figure 19 shows a simple network of PCs in which RU A has sent a search mode message to RU B. The arrows on the communication lines in the figure show the direction that messages might flow during the search. The behavior of any actual network cannot be predicted, however, since the difference between rejection and acceptance of a particular copy of the message at a given PC is a function of the detailed timing history at that PC. For instance, the PC marked with an asterisk in Figure 19 is bound to receive, almost simultaneously, two copies of the message; the first of the two messages to get the attention of the I/O processor will be accepted.

The arrows shown inside each PC in Figure 19 represent what the operating system would decide about the direction a message should be sent in order to reach RU A. All messages for RU A will be sent in those directions.

2. Directed Mode

When the message arrives at RU B, an acknowledgement is returned to RU A. Since the route back now is known, the message will travel back using the second mode (the directed mode) . The same route notes are made by each PC dealing with the return message. Therefore, the direction in which a message might be sent to RU is known by each of the PCs on the direct return route. It is important to note that no PC knows the route to any RU--all that is known is the direction that has worked before.

3. Default Mode

On occasion, a message will be moving through a directed route and arrive at a place in the network where the direction to the target is

not known. This causes the third mode of transmission, the default mode, to be invoked. In the default mode, the message is sent to all PCs which have not yet seen the message (as in the search mode). When the message reaches the target successfully, the acknowledgement is sent back via the directed mode. Once a message drops into the default mode, it remains in that mode even though it may encounter route information along the way.

4. Broadcast Mode

The fourth mode of transmission is called broadcast mode, and differs from the other modes in that it is used for system-related messages sent by a PC to all other PCs. The name strongly suggests this behavior. As in the search and default modes, a broadcast message is sent out all I/O ports except the one by which it arrived.

5. Local Acknowledgement

Local acknowledgements are sent for each message received by a PC. A local acknowldegement is a message that travels no further than the directly connected neighbor. It is a "handshake" function which indicates that the receiving center has accepted the message. If a local acknowledgement is not received, the sending center infers that the message is not proceeding successfully. In that case, the supervisor processor is notified that a neighbor is possibly faulty, and the message is re-sent in default mode, if appropriate.

6. Major Acknowledgement

Major acknowledgements are sent through all intervening PCs back to the source along the learned route. Should the learned route be interrupted by the time the acknowledgement is returning, the acknowledgement reverts to default mode in exactly the way previously described. A message stating that the acknowledgement had to change to default mode is sent back to the PC containing the source RU to indicate that the route to the destination has been destroyed. In this situation, a new route should be sought immediately in case it should be necessary to call a backup RU into action. The problem here is that in this situation, the RU at the destination cannot be counted

upon, therefore extra attention must be given by the operating system to make sure that it is still functioning or that it still can be reached.

V FAULT TOLERANCE AND THE SUPERVISOR PROCESSOR

The term "fault tolerance" as used in this report refers to a characteristic behavior of the network computer which, after a failure of some kind, seeks to restore correct functioning of the network and to generate correct results in the output data stream. Fault tolerance is a hierarchical design problem. The level of fault tolerance and the detailed behavior during recovery must be tailored to the needs of the application. Consideration must be given to cost of high degrees of fault tolerance, as well as to the expected failure mechanisms.

Figure 20 shows the layered structure upon which fault tolerance (reliability) is built. The first engineering level is that of producing reliable hardware. Literature and the design rules for creating reliable hardware against many specific threats is well developed. With such hardware, a homogeneous architecture such as CHAMP may be constructed to take advantage of the presence of many identical units which can replace one another in the case of failure. In order to manage such a homogeneous architecture, there is a need for a fault-handling operating system. Such an operating system is the subject of this section of the report.

In addition to the system requirements mentioned above, it is possible to impose yet other, more stringent requirements which can be met by elaborate software schemes. Such schemes map naturally onto a network such as CHAMP, and are exemplified by the Software Implemented Fault Tolerance (SIFT) algorithm developed by SRI.

To achieve a high degree of fault tolerance, diagnostics are most important. In fact, since exhaustive diagnostics are not possible (they require almost infinite time to execute), a balance must occur wherein a portion of processing time is devoted to diagnostics. The primary importance of diagnostics is that fault-recovery algorithms depend on the existence of valid and functioning hardware when new resources are needed to replace those that are judged faulty.

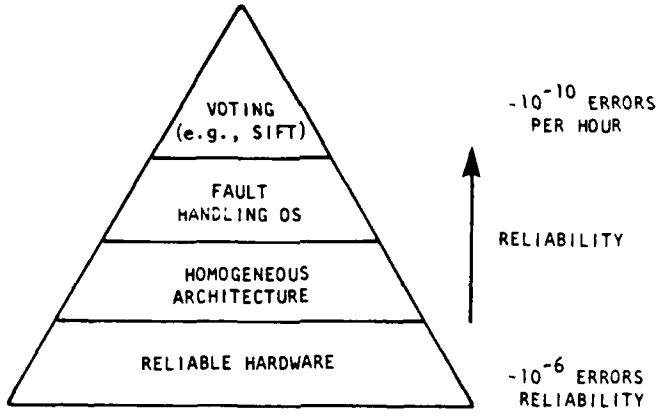


FIGURE 20 DIAGRAM OF FAULT TOLERANCE HIERARCHY

As already mentioned, the CHAMP architecture is intended to make extensive use of diagnostics. It is reasonable to assume that 50% of the processing resources should be available for diagnostics and that the diagnostics be relevant to the task being performed.

In the CHAMP system, basic fault recovery is managed locally by the supervisor processor, with help from the I/O processor. Each PC is responsible for knowing the status of its own resources and the resources of its directly connected neighbors. This is achieved on the lowest level by deliberate diagnostics, which are constructed of simple routines which exercise all or nearly all of the PCs major resources. Other diagnostics may be constructed of subelements of applications routines which are normally found running in the processors.

Figure 21 shows a subsection of the network wherein five processing centers are running together. The central processor in this figure has checked itself and found itself functional. It has, in addition, checked neighbors 3 and 4 and found that they give valid responses while neighbor 2 has returned an unsatisfactory response and is therefore judged faulty. In this rudimentary way, a file of the health of each neighbor is maintained by each PC.

Once a faulty PC is found, more detailed diagnostics are run in an effort to localize the site of the failures into one of the subelements of the PC. Even if a neighbor is known to have a faulty task processor it still might be possible or even desirable to use the communications portion of that neighbor in order to send messages to a destination RU on the other side of that neighbor. Any technique that reduces the possibility of fragmentation of the network into disconnected islands is helpful.

Spare resources are called into service if acknowledgement is received for a data message or if the expected input data message is not received. For example, suppose RU W has sent data to another RU named X. At the same time the message is sent, a counter is started in the PC containing RU W, indicating that a reply is expected within a

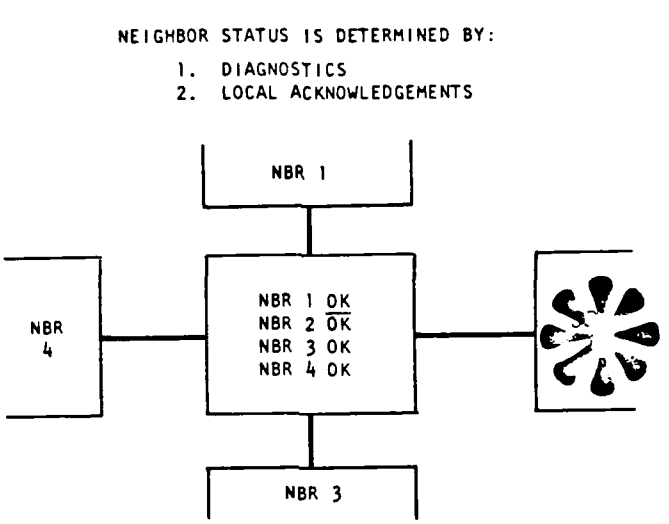


FIGURE 21 DIAGRAM SHOWING LOCALLY MANAGED FAULT DIAGNOSTICS AMONG NEIGHBORS MANAGED IN THE NETWORK

definite time. If at the end of this time a reply is not received, the supervisor processor managing RU W broadcasts a message that the first backup copy for RU X should be started and that the backup data should be augmented by the present message data in order to perform the startup.

During normal operation of the network there are several backup copies for each RU that is active. Each of these backup copies receives checkpoint data from which to start up processing in case they should be called upon to do so. The checkpoint data are sent to the backup copies by the main or the active copy of that RU. Our analysis of the avionic functions of navigation and flight control indicate that the additional message traffic implied in this process, as detailed in Section VI, is acceptable even in a high-damage situation. This would be more accurate if six I/O ports were used instead of the four assumed by this study.

Each backup copy of a RU of code is treated like every module of code, except that it is held inactive by the owning supervisor processor until the activation request is received. The only other difference in treatment is that duplicate backup copies may not reside together in the same processing center; this is handled by the load-levelling algorithm.

Load levelling (load equalization) is managed by each supervisor processor independently. One version of the load-levelling algorithm moves modules of code from one PC to another in order to decrease the sum-squared load. This is an example of the core of the locally controlled and locally effective load-management portion of the operating system.

Figure 22 illustrates this load-levelling algorithm in action. Four PCs are shown connected, each having the processing load shown. The PC in the middle has a load of 3; each of its neighbors has a load of 2; another has a load of 1. The stability theorum permits various techniques for assessing the neighbors load situation; however, the one shown here is the sum-squared load. In this example, the situation

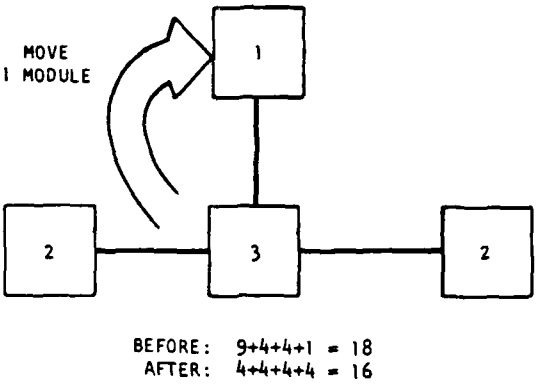


FIGURE 22 ILLUSTRATION OF LOAD-LEVELLING ALGORITHM IN OPERATION

presents a sum-squared load of $3^2 + 2^2 + 2^2 + 1^2 = 18$ for the squared load of the group shown. The supervisor processor in the central PC constantly postulates the movement of load units to its neighbors; in this case the postulate considers moving one load unit to the neighbor who presently has a load of one. This adjustment would give each of the three neighbors a load of two and the central unit a load of two. The resulting sum-squared load for the group is 16. Since 16 is less than 18, the supervisor processor judges that the move would be desirable; therefore, it sends one load unit to the selected neighbor.

It should be evident that it is not necessary to consider all connected neighbors in order to postulate possible moves. This algorithm only requires that any given pair of PCs move toward a lower sum-squared load as a result of any given move. This simple algorithm does not necessarily result in the optimum load structure for the network when viewed globally. However, it does result in a stable and improved position after a small number of relocations.

The same procedure is invoked when damage occurs due to the activation of backup units which create temporary load peaks in portions of the network. As was mentioned before, additional constraints must be placed on the algorithm to ensure that duplicate backup modules do not reside in the same PC, and that related RUs do not tend to drift into unrelated neighborhoods. One way to help disperse duplicate copies of backups is to assign a penalty function (e.g., 2) to the load of an RU if a postulate is formed which places it next to an identical copy.

The load-levelling procedure is essential for any computing network which needs to survive hostile action A significant breakthrough obtained in this contract is the analytical proof that there exists a class of stable, locally controlled and locally effective load-management algorithms to perform the desired function in a finite and short period of time. A more detailed description of this class of algorithms and the proof of stability is given in Appendix D.

It is more important to note neither the message routing algorithms nor the load-levelling algorithms contains any assumption as to the geometric pattern in which the PCs are connected. The operating system is effective for any connectivity, and so there is great design freedom for the engineer. In addition, this characteristic allows accidential reconnection of the network into another configuration at the field level without any serious effect on the functioning of the computer. Thus, a service shop might unintentionally scramble some of the communication links without creating failures or improper operation.

Recovery from Massive Failures

Designing for recovery from massive damage has three principal requirements:

(1) Maintain sufficient spare copies of software so that at least one usable copy remains after damage.

(2) Detect the damage and reconfigure the network of RUs according to priorities.

(3) Protect against the possibility of cutting off communications between the network and corresponding sensors and activators.

In the first case (maintenance of sufficient spares) one must define sufficiency. In a military situation, sufficiency may mean that within some probability (say 95%) damage of up to a certain limit (say 50%) will not prevent recovery.

As an example, the CHAMP network was sized using this criterion, as detailed in Section III, for each RU. There are four copies of each RU in the system--one RU actively processing and three backup copies. If the duplicate copies of RUs are randomly distributed in 20 PCs and if 50% (i.e., 10) of the centers are destroyed, then the probability that exactly four of the copies of a given RU are destroyed is:

$$P_4 = \frac{\binom{4}{4}\binom{16}{6}}{\binom{20}{10}} = 0.04$$

where $\binom{X}{Y}$ is a binominal coefficient which is equivalent to*

$$\frac{X!}{Y!\ (X-Y)!}$$

More elaborate considerations are involved if the requirement is to be that no RU is at the 95% confidence level to suffer elimination at the 50% damage level. These are beyond the scope of this report, but the procedures involved are common design practice.

If the quantity of hardware is not a problem in the Survivable Avionics Computer System, or if there are functions considered so vital that under no circumstances are all copies of certain RUs to be eliminated, then one need only include more copies than can be eliminated by the postulated damage. As an example, at the 50% damage level this criterion requires that there be 11 copies of each vital RU in a 20-PC network.

The second requirement in desinging the system for recovery is that the network of RUs must be reconnected and the tasks restarted after damage. The algorithms described in this section are a starting point and were developed as a result of the damage recovery simulation study done under this contract; therefore, they are not included in the simulation. More work is needed in this area.

Each active (primary) RU maintains a clock (the data arrrival clock = DAC) on each item of input data it normally receives. The RU also knows from which RU the data are supplied. Note that neither the active RU nor the PC on which it is running knows where the source RU is only its name and, at most, the direction from which the data are usually received. The DAC is set to "time-out" (a design parameter) shortly after the data are expected, but, if possible, well before the data are needed. These time limits must be provided by the programmers, although there may be a way to generate them automatically.

---

*Feller, C.F., "Probability Theory and Its Applications," New York: Wiley, p33 (1950).

Should the DAC time-out before the data arrive, the supervisor processor broadcasts a message to all backup copies of the presumed missing RU. All backup copies respond with a message containing the required data obtained from their roll-back files (they have been following the progress of the computation at specified intervals). The first such message to arrive is acknowledged and the others are ignored; and thereby, only one backup is activated. The others go back to watchful sleep.

The activated backup RU then broadcasts throughout the network looking for its sources of data. Up to this point it has been receiving checkpoint data, but possibly only from the former primary RU. Now it needs to be connected to the necessary source RUs. All source RUs (active and backup) answer. The active source RUs are acknowledged if they reply, and if an active copy does not reply, then the first backup copy to reply is acknowledged.

Thus the virtual network is reconnected starting with the RU whose DAC has timed-out, and the process continues upstream through all data sources in turn. If the reconfiguration chain is a long one, the time taken to reconfigure and to bring roll-back data to currency could be too long for an effective restart. The trade-off is to maintain cycle-by-cycle current checkpoint data in all backup RUs in a chain when reconfiguration time will be limited. The steady-state message overhead would be increased, but the recovery time would be greatly reduced. Additionally, priority tags are attached to all reconfiguration requests and acknowledgement messages so that communications resource conflicts can be resolved immediately in favor of the highest priority requestor.

To help clarify this algorithm, consider the simplified example in Figure 23.

RUs A - E are positioned together with backup copies, A' through E', in the network of PCs (1 through 8). The normal communication chain for the computation illustrated is A B C D E. Suppose PC6 and PC4 are destroyed at the same time, and that the DAC in E is the first to time-out. The situation is now as shown in Figure 24.

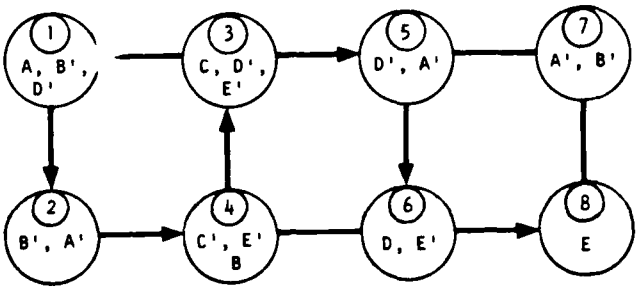


FIGURE 23 DISTRIBUTION AND COMMUNICATIONS FLOW PRIOR TO SYSTEM DAMAGE

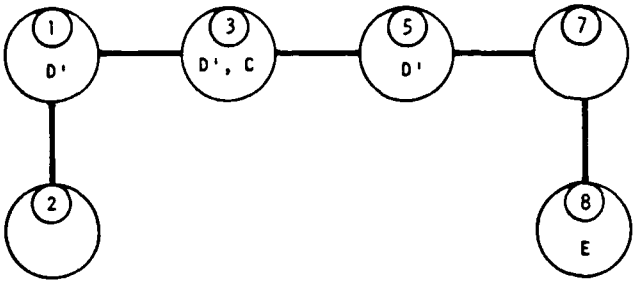


FIGURE 24 REESTABLISHED COMMUNICATIONS BETWEEN D AND E AFTER SIMULTANEOUS DAMAGE

E sends a request for the nearest D' to reply. The one in PC 5 will, most likely, respond first. When D' in 5 receives the go-ahead acknowledgement from E, it requests C to respond. C still exists in 3, and so C just redirects all traffic to the new copy of D in 5 instead of the old primary D that was in 6. Now a subtle point--C is a primary RU and so it still may believe itself connected to its source. When the DAC in C times-out, the process begins again, carrying the reconnection all the way back to the source.

If the last RU in a computational chain is destroyed (e.g., the one connected to an actuator) there may be no active DAC to notice the problem. In such cases it is desirable to make all copies of an output RU active copies, or at least to require all output DACs to be active.

In cases when it is possible, it may be necessary to reload the network following massive failures. This is a special case, however,

and would require extra time to assess the network and load RUs according to their priorities. This case may also result in an intolerable delay in the data flow. Reloading from an external archive is a design trade-off that should be used only when the alternatives are more undesirable.

The recovery algorithm described here is not a complete one, and it raises some issues that need to be explored further. However, it serves to illustrate the direction in which research might be carried.

The third consideration in designing the system for recovery is that the network must remain connected, at least to the extent that all RUs in a given computational group (i.e., navigation) remain connected to each other and to the corresponding sensor and actuator systems. This is a communications design problem and is sensitive to the way in which remote aircraft systems are joined by busses or links. One useful practice might be to locate as many of the RUs as possible physically close to the sensor or actuator needed. In this way, if the actuator is destroyed, there is no additional penalty attached to losing the associated computational resources. If co-location is not possible or desirable, careful assessment must be made of the vulnerability of planned communications links.

VI SIMULATION

Simulations show that the CHAMP network has a capability for survival. Two sequential simulations were constructed and run on SRI's KL-10 computer to demonstrate the vital functions of communications and load levelling. Because of the complexity of these algorithms and the elaborate failure modes possible, the simulation study was divided into two portions in order to pinpoint the recovery behavior more accurately. This approach has drawbacks shared by all but the most exhaustive and time-consuming simulations in that assumptions made in order to produce tractable simulations can influence the results of the simulation. This is discussed in more detail later.

Results of an initial, simple simulation for each of the two functions are reported in Appendices C and D. This section deals with more elaborate simulations using the characteristics of the flight control and navigation functions previously defined in Section III.

The principal results of these simulations are: (1) locally controlled load-levelling algorithm remains stable even in the presence of significant failures--in fact, more than 50% failures, and (2) the communications algorithm continues to function and provide communication facilities in the event of the same amount of damage. In both cases it is important that the network be designed with sufficient connectivity to ensure that damage does not divide the network into fragments, where one portion is connected to a sensor and another is connected to the corresponding equation. For that reason, it is probably desirable to use PCs with more than four I/O ports as previously mentioned. However, the simulations reported here with four I/O ports tend to accentuate any troublesome behavior which might be experienced.

Several load-levelling simulation runs were made in the course of this study. Two of these runs are reported in this section.

The 44 RUs comprising navigation and flight control were distributed in the network by the load-balancing algorithm in a start-up

operation. That is to say, all 44 RUs were loaded into PC 1, and the operating system distributed the RUs according to the sum-squared load algorithm. Failures were induced sequentially by removing one processing center at a time from the network. At this level of simulation, the backup modules for each RU were assumed to be one step away from the primary modules. Secondary backups were located further away; however, since failures were induced sequentially, the neighboring backup was always the one activated in the recovery process.

Figure 25 shows the load situation at the beginning of the run, and Figure 26 shows the activity that occured following the failure of PC 1. Backup modules for the now-inoperative RUs were started in the three neighbors of PC 1, and new backup copies were created in PCs adjacent to the neighbors of PC 1. Following the recovery and restart of the backup modules, the load was releveled and, in this case, required four steps (Figure 27) to achieve a load balance. Figure 28 illustrates the load situation after the failure. This procedure was continued until more than half of the PCs were taken from the network.

Figure 29 shows two graphs of the number of steps required to achieve restabilization as a function of the number of bad or failed PCs in the network. One of the graphs shows a large peak following the failure of two PCs. The reason for this anomoly is that the PC involved in the second failure contained one of the largest RUs in the network. The load represented by that RU was 19, whereas most of the other RUs represented a load of 5 or less. Two runs did not establish a trend, but the two graphs show that relatively few steps were required for the subject avionics modules, even in the presence of a large number of failures.

Figure 30 is a diagram of the situation when 50% damage had occured. The tenth PC removed was PC 18. Because of the geometry involved, the backups for PC 18 were activated in a single neighbor and all of the new backups were created in another single PC. Figure 31 shows the load situation following recovery of the RUs from failed PC 18. Thereafter, 19 steps were required to rebalance the load. The steps involved in the re-balancing process are shown in Figure 32.

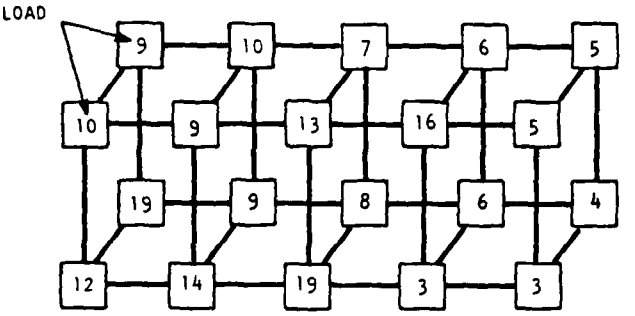


FIGURE 25 DIAGRAM OF LOAD SITUATION AT BEGINNING OF RUN

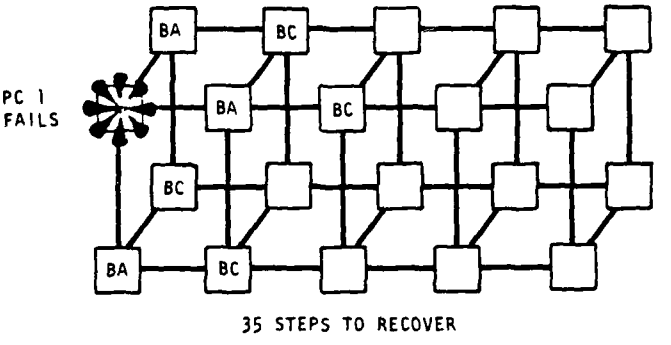


FIGURE 26 DIAGRAM OF ACTIVITY WITHIN NETWORK AFTER FAILURE OF PC

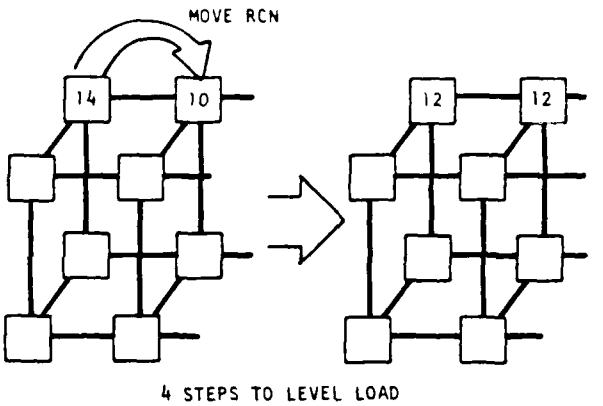


FIGURE 27 DIAGRAM OF STEPS REQUIRED TO RELEVEL LOAD AFTER FAILURE OF PC

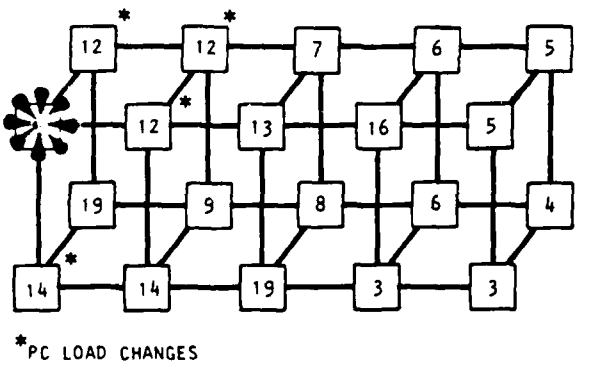


FIGURE 28 ILLUSTRATION OF LOAD SITUATION AFTER LOAD LEVELLING

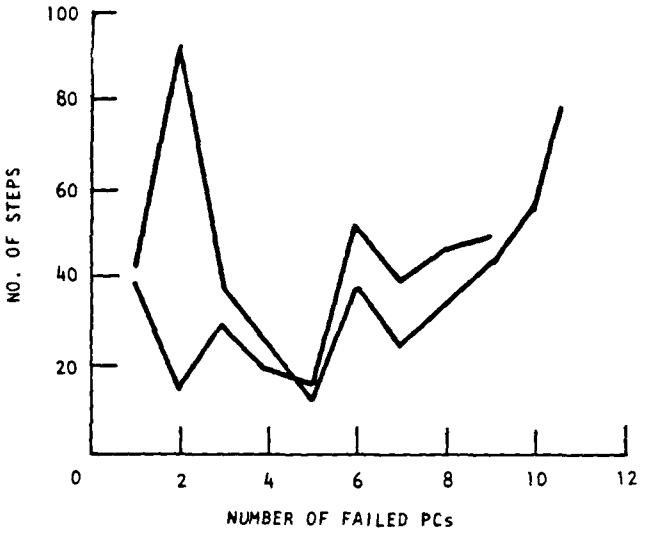


FIGURE 29 DIAGRAM OF NUMBER OF STEPS REQUIRED TO RESTABILIZE NETWORK IN RELATION TO NUMBER OF FAILED PCs

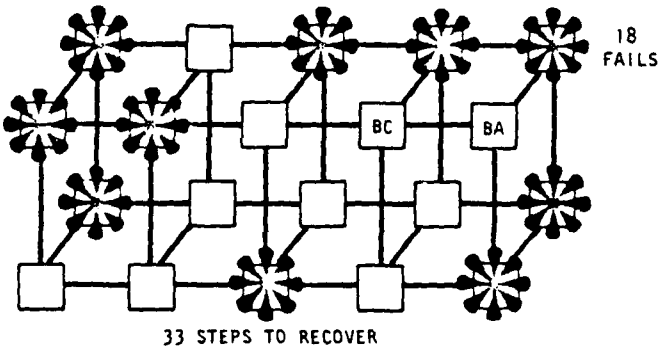


FIGURE 30 DIAGRAM OF NETWORK ACTIVITY AFTER A FAILURE THAT CAUSES 50% SYSTEM DAMAGE

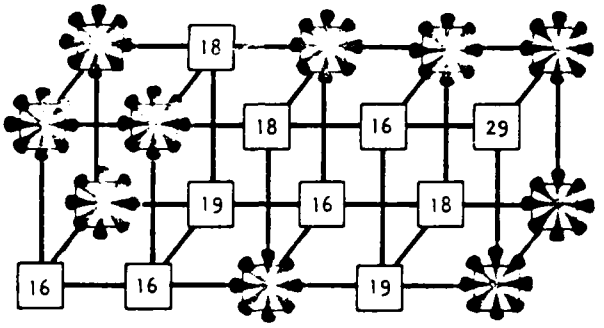


FIGURE 31 DIAGRAM OF LOAD SITUATION AFTER RECOVERY (BUT BEFORE LOAD LEVELLING) OF THE RUs FROM PC 18 FAILURE

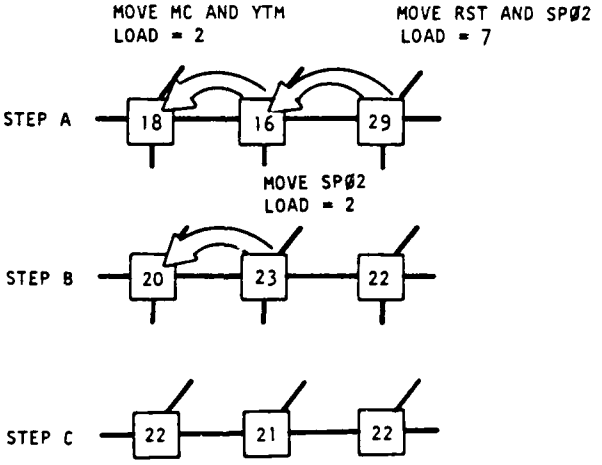


FIGURE 32 DIAGRAM OF STEPS INVOLVED IN REBALANCING THE LOAD

The results shown here are typical of those we experienced in the simulations. Failures were selected at random, with no particular bias or preference except that care was taken not to fragment the network into unconnected pieces.

Table 11 is a detailed summary of the sequence of events in one of the load-balancing-algorithm simulation runs.

Of the two major simulation runs for the load-levelling algorithm, 110 steps was the maximum number of steps needed to recover from a failure. Usually, the most steps occured after 50% damage. The timing of these recovery steps is of crucial importance in some applications; depending on the normal update interval for each RU, the reconfiguration steps could interfere with the normal operation of some RUs. For this reason, priority must be given to the reactivation of RUs which must perform in a short period of time. These RUs should be moved first and started first in the sequence that follows a reconfiguration.

Turning now to the communications algorithms, several simulations were run at various levels of damage in the communications network. The results showed that the communications situation in the avionics network is much less demanding than the load-balancing algorithm. One reason for this is the relatively straightforward behavior of the message routing algorithms, the behavior of which can be predicted ahead of time for a sequential-style simulation. Another reason is the relatively low communications load imposed by the navigation and flight control RUs. The results of one run are discussed in the following paragraphs and serve to illustrate the modes of behavior of CHAMP communications.

Figure 33 shows a situation that was simulated in which the first message sent in the newly load-balanced network is from RU YTM to RU RCD of the flight control function. Since this was the first message sent, the route was not known and the message proceeded in search mode in the directions shown by the arrows. As was described in Section IV, the acknowledgement can and does return in directed mode along the learned route as shown in Figure 34. The knowledge of the route to RU

Table 11

RU LOAD DISTRIBUTION AS FAILURES INCREASE TO 50 PERCENT

	Processing Center																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Load =	10	9	19	12	9	10	9	14	13	7	8	19	16	6	6	3	5	5	4	3
RUs =	MC	QP26	SPØ3	PIN	SPØ5	ECP	PST	ABD	SP21	QP25	PSM	QPØ4	RM	RCD	YRT	AHD	RST	PCN	PHD	SPØ2
	YTM	LAC		ACD	SP2Ø	ECD	SPØ6	SPØ4		SPØ7				QP21		SPØ8		QPØ3	SP12	SP11
Ø PCs	PTM	NAC		LIN										QP2Ø				QP15	QPØ2	
Failed	RCN													QP16						
	RAC																			
	GS																			
	QP4Ø																			
Load =	Ø	12	19	14	12	12	9	14	13	7	8	19	16	6	6	3	5	5	4	3
RUs =	Ø	QP26	SPØ3	PIN	SPØ5	ECP	PST	ABD	SP21	QP25	PSM	QPØ4	RM	RCD	YRT	AHD	RST	PCN	PHD	SPØ3
		LAC		ACD	SP2Ø	ECD	SPØ6	SPØ4		SPØ7				QP21		SPØ8		QPØ3	SP12	SP11
1 PC		NAC		LIN	GS	RCN								QP2Ø				QP15	QPØ2	
		QP4Ø		PAC	YTM									QP16						
Failed		RAC			MC															
		PTM																		
Load =	Ø	Ø	19	16	12	11	9	16	13	9	8	19	16	7	6	3	5	8	6	5
RUs =	Ø	Ø	SPØ3	PIN	PSØ5	ECD	PST	ABD	SP21	RCN	PSM	QPØ4	RM	QP25	YRT	AHD	RST	QPØ3	PHD	SPØ2
				ACD	SP2Ø	LAC	SPØ6	SPØ4		ECP				SPØ7		SPØ8		RCD	PCN	SP11
2 PCs				LIN	GS	NAC		PAC										QP2Ø	QP15	SP12
				QP4Ø	YTM													QP21		QPØ2
Failed				QP26	MC													QP16		
				RAC																
				PTM																
Load =	Ø	Ø	Ø	16	12	11	19	16	13	9	9	19	16	8	8	3	5	10	5	5
RUs =	Ø	Ø	Ø	PIN	SPØ5	ECD	SPØ3	ABD	SP21	RCN	PST	QPØ4	RM	QP25	PSM	AHD	RST	RCD	PCN	SPØ2
				ACD	SP2Ø	LAC		SPØ4		ECP	SPØ6			SPØ7		SPØ8		QP2Ø	YRT	SP11
3 PCs				LIN	GS	NAC		PAC						QPØ3				QP21		SP12
				QP4Ø	YTM													QP16		QPØ2
Failed				QP26	MC													PHD		
				RAC														QP15		
				PTM																
Load =	Ø	Ø	Ø	16	12	13	19	16	13	Ø	9	19	16	13	9	3	5	11	5	5
RUs =	Ø	Ø	Ø	PIN	SPØ5	ECD	SPØ3	ABD	SP21	Ø	PST	QPØ4	RM	SPØ7	PSM	AHD	RST	RCD	PCN	SPØ2
				ACD	SP2Ø	LAC		SPØ4			SPØ6			ECP	QP2Ø	SPØ8		QP21	YRT	SP11
4 PCs				LIN	GS	NAC		PAC										QP16		SP12
				QP4Ø	YTM	RCN												PHD		QPØ2
Failed				QP26	MC													QP15		
				RAC														QPØ3		
				PTM														QP25		

Table 11 (Concluded)

RU LOAD DISTRIBUTION AS FAILURES INCREASE TO 50 PERCENT

	Processing Center																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Load = RUs = 5 PCs Failed	0 0	0 0	0 0	16 PIN ACD LIN QP40 QP26 RAC PTM	12 SP05 SP20 GS YTM MC	13 ECD LAC NAC RCN	19 SP03	16 ABD SP04 PAC	13 SP21	0 0	9 PST SP06	0 0	16 RM	13 SP07 ECP	9 PSM QP20	19 QP04	5 RST	11 RCD QP21 QP16 PHD QP15 QP03 QP25	5 PCN YRT	8 SP03 SP11 SP12 QP02 SP08 AHD
Load = RUs = 6 PCs Failed	0 0	0 0	0 0	16 PIN ACD LIN QP40 QP26 RAC PTM	12 SP05 SP20 GS YTM MC	13 ECD LAC NAC RCN	19 SP03	16 ABD SP04 PAC	13 SP21	0 0	9 PST SP06	0 0	16 RM	13 SP07 ECP	9 PSM QP20	19 QP04	11 RST SP11 SP02 PCN	11 RCD QP21 QP16 PHD QP15 QP03 QP25	0 0	7 SP12 QP02 SP08 AHD YRT
Load = RUs = 7 PCs Failed	0 0	0 0	0 0	16 PIN ACD LIN QP40 QP26 RAC PTM	12 SP05 SP20 GS YTM MC	13 ECD LAC NAC RCN	19 SP03	16 ABD SP04 PAC	13 SP21	0 0	10 PST SP06 QP20	0 0	16 RM	0 0	14 PSM SP07	19 QP04	14 RST SP11 SP02 PCN RCD	15 QP21 QP11 PHD QP15 QP03 QP25 ECP	0 0	7 SP21 QP02 SP08 AHD YRT
Load = RUs = 8 PCs Failed	0 0	0 0	0 0	16 PIN ACD LIN QP40 QP26 RAC PTM	0 0	18 ECD LAC NAC RCN SP20	19 SP03	16 ABD SP04 PAC	18 SP21 GS SP05	0 0	10 PST SP06 QP20	0 0	18 RM YTM MC	0 0	14 PSM SP07	19 QP04	14 RST SP11 SP02 PCN RCD	15 QP21 QP16 PHD QP15 QP03 QP25 ECP	0 0	7 SP12 QP02 SP08 AHD YRT
Load = RUs = 9 CPs Failed	0 0	0 0	0 0	16 PIN ACD LIN QP40 QP26 RAC PTM	0 0	18 ECD LAC NAC RCN SP20	19 SP03	16 ABD SP04 PAC	18 SP21 GS SP05	0 0	16 PST SP06 QP20 SP07	0 0	18 RM YTM MC	0 0	14 PSM QP02 YRT SP08 AHD	19 QP04	15 RST SP11 SP02 PCN RCD SP12	15 QP21 QP16 PHD QP15 QP03 QP25 ECP	0 0	0 0
Load = RUs = 10 PCs Failed	0 0	0 0	0 0	16 PIN ACD LIN QP40 QP26 RAC PTM	0 0	18 ECD LAC NAC RCN SP20	19 SP03	16 ABD SP04 PAC	22 SP21 GS SP05 SP02 YTM MC	0 0	16 PST SP06 QP20 SP07	0 0	21 RM RST	0 0	18 PSM QP02 YRT SP08 AHD	19 QP04	22 SP11 PCN RCD SP12 QP16 QP15 QP03 QP21	0 0	0 0	0 0

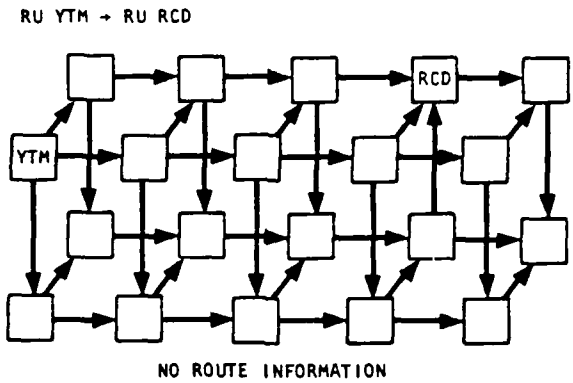


FIGURE 33 DIAGRAM SHOWING THAT THE FIRST MESSAGE SENT INTO THE LOAD-BALANCED NETWORK IS TRANSMITTED IN SEARCH MODE

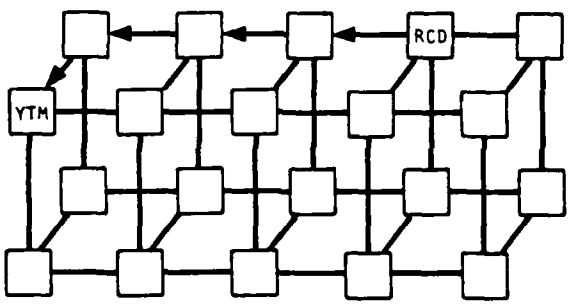


FIGURE 34 DIAGRAM OF ACKNOWLEDGEMENT RETURNING IN DIRECTED MODE ALONG THE LEARNED ROUTE

RCD was used by other modules wishing to communicate with that RU (Figure 35) in which RU LAC is communicating with RU RCD. Nothing very spectacular occured as damage levels were increased in this simulation, except that the messages switched from search mode to directed mode to default mode with predictable regularity.

One situation, however, requires special attention: the near segmentation of the network into two pieces as shown in Figure 36. If one of the RUs located in the smaller portion of the network attempts to send a message to a RU located in the larger half of the network, it could find the old known routes are blocked. In one simulation run this did occur. Figure 37 shows the situation at the critical time. When a blockage was detected by two of the PCs , a warning message was returned to the supervisor of the PC originating the message. This warning caused the originating PC to re-send RU LAC in the default mode to RU RCD. Figure 38 shows the situation that occured when LAC was re-sent in default mode. Route information which may be available along the way was ignored, since damage may have caused these routes to be wrong and could prevent the message from arriving at the intended destination. As before, the acknowledgement returns along the route that was learned while the message propagated in default mode to its destination (Figure 39).

This warning feature was developed in order to prevent a message from entering a blind alley and then being unable to retrace its steps since the algorithm for broadcast and default modes requires a message that already has been seen to be rejected. Further details for this algorithm and the need for it are reported more fully in Appendix C.

In summary, one key to survival of the network is that it must remain connected (Figure 40) in the sense that all parts of any given external section must be connected to the same surviving fragment. A hardware design to enable the network to remain connected under any assumed threat condition is an engineering problem of major importance. This design procedure involves trade-off factors similar

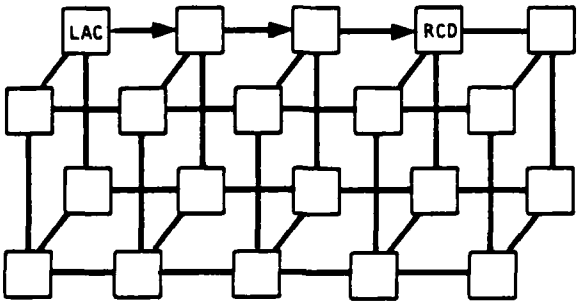


FIGURE 35 DIAGRAM SHOWING THAT ROUTE KNOWLEDGE IS USED BY OTHER MODULES WISHING TO COMMUNICATE WITH AN RU

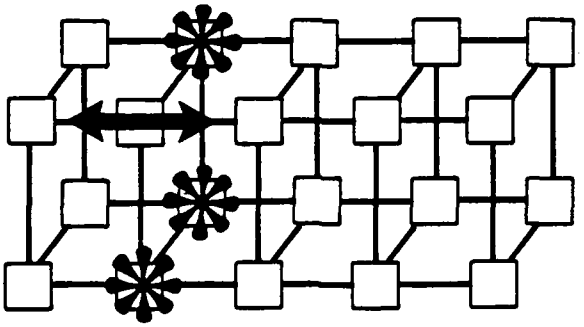


FIGURE 36 DIAGRAM OF NETWORK DAMAGE CAUSING RESTRICTED COMMUNICATIONS

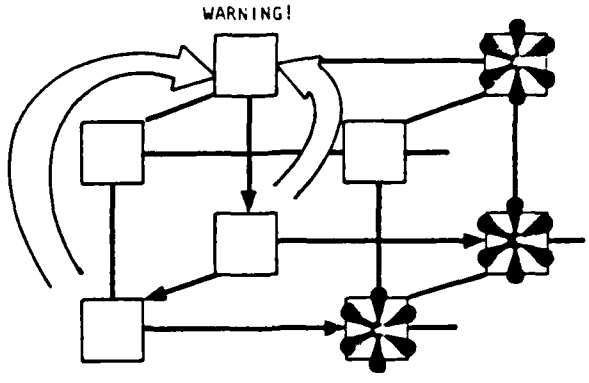


FIGURE 37 DIAGRAM SHOWING WARNING MESSAGE SENT TO RU MESSAGE ORIGINATOR INDICATING RU MESSAGE MAY ENCOUNTER BLOCKAGE IN NETWORK

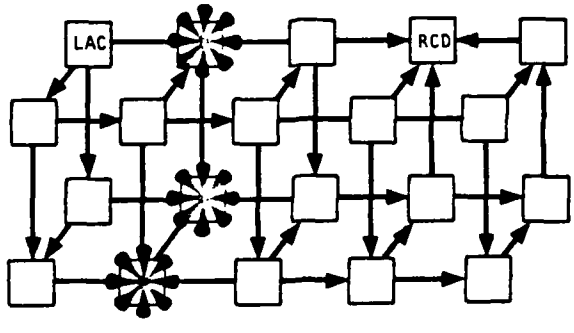


FIGURE 38 DIAGRAM OF MESSAGE MODE CHANGING AFTER DAMAGE TO ENSURE THAT ALL PATHS ARE TRIED

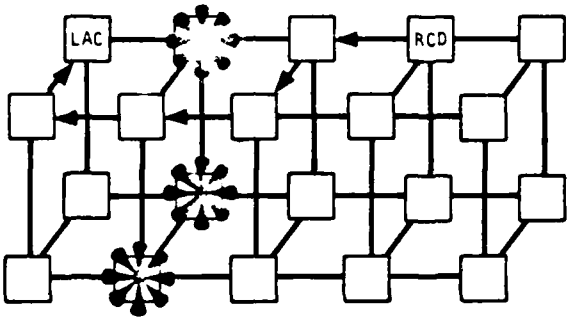


FIGURE 39 DIAGRAM SHOWING ACKNOWLEDGEMENT RETURNING ALONG LEARNED ROUTE AFTER DAMAGE

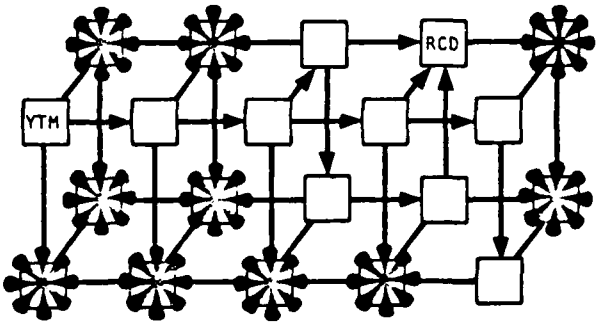


FIGURE 40 DIAGRAM SHOWING IMPORTANCE OF NETWORK REMAINING CONNECTED (EXTERNAL SECTIONS MUST REMAIN CONNECTED TO SURVIVING FRAGMENT)

to those encountered by airframe designers who must consider the design and location of protective armor for the pilot and for the aircraft structure; there are both advantages and disadvantages to any designed feature.

Tables 12 and 13 summarize the peak communications demand presented by each RU for the navigation and flight control functions. The total peak load for both functions is 13,737 words/second if every RU were to require communication over a single link at one time. This is less than 1/4 the capacity of a simple 1-MHz transmission link. Figure 41 shows the communications load situation that would exist on a single 1-MHz communications line if all messages were to require that line at the same time. Even assuming that the overhead burden is three times the message load for navigation and flight control, there is 6.4% spare capacity on a 1-MHz message link .

This analysis shows that for the case of navigation and flight control under heavy damage conditions where a single link may be connecting two "islands" of PCs, there is generous capacity for the required comunications, especially since future transmission links are likely to be faster than 1 MHz. It is for this reason that the communications simulation showed less demanding environment than the load management simulation.

The algorithms are straightforward and their behavior is predictable, especially in a sequential situation such as the one we constructed. Some possible problems not disclosed by this simulation are related to resource conflict situations. This situation involves two or more messages in time-critical contention for the same resource. This contention would cause the creation of a learned route different from the ones given in our simulations as a result of search mode communications. Such differences could cause more circuituous routes through the network from RU to RU. However, for the navigation and flight control functions, the extra delay implied by longer paths is not a significant factor.

A much more effective assessment of the algorithmic behavior can be achieved with a hardware simulation, because the effects of the

Table 12

NAVIGATION MESSAGE REQUIREMENTS

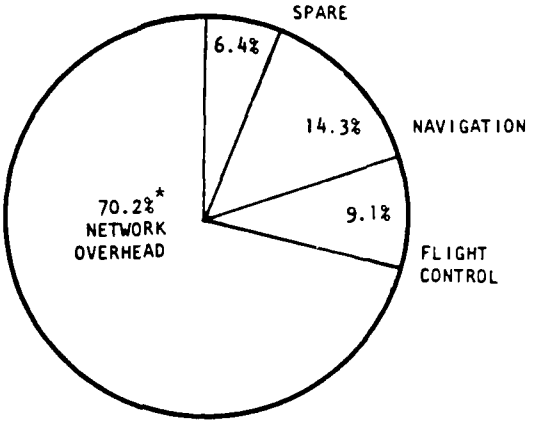
RU	Messages	Period (ms)	Peak Rate (words/s)
SP02	10	256	39
SP03	16	32	500
SP04	10	128	78
SP05	12	256	47
SP07	59	32	1843
SP08	20	1024	20
SP11	58	128	453
SP12	40	1024A	39
SP21	10	128	289
QP02	11	128	86
QP03	35	1024A	34
QP04	125	32	3906
QP15	2	1024A	34
QP16	74	128	578
QP20	10	1024A	10
QP21	15	128	117
QP25	7	1024A	7
QP36	9	128	70
QP40	24	128	188
TOTAL			8385

Note: All asynchronous messages (marked with "A" in period column) were assigned a period of 1024 ms.

Table 13

FLIGHT CONTROL MESSAGE REQUIREMENTS

RU	Messages	Period (ms)	Peak Rate (words/s)
RM	51	12.5	4080
MC	9	100	90
AHP	1	50	20
PAC	1	25	40
ECP	1	25	40
RAC	1	25	40
GS	10	100	100
PHD	1	25	40
PTM	1	50	20
NAC	1	25	40
PST	2	25	80
PCN	1	25	40
PSM	2	12.5	160
PIN	1	12.5	80
ECD	1	12.5	80
RST	1	25	40
ACD	1	25	40
YTM	1	25	40
LAC	2	25	80
LIN	1	25	40
RCN	1	25	40
YRT	1	25	40
ABD	1	50	20
RCD	1	25	20
TOTAL			5329



*ASSUMES OVERHEAD IS 3X MESSAGE LOAD

FIGURE 41 ILLUSTRATION SHOWING WORST-CASE TRAFFIC LOAD ON A SINGLE 1-MHz COMMUNICATIONS LINK

sequential computer simulation would be removed from the results an contention situations could naturally arise. Particularly, the trou blesome effects thus avoided are time-slice effects, in which event are forced into an artificial order governed by the order in which th lines of code consider the PCs and the messages among them. In hardware network, it would be possible for messages to arrive simul taneously and create a timing or resource conflict, or an apparent lo jam, which does not occur in a sequential simulation. More investiga tion of this problem is needed.

SRI is undertaking, on internal funds, a simple hardware demonstration which is intended to show the functioning of the load-leveling and communications algorithms in the absence of significant user-module processing. Although this is a useful step, it still lacks the timing and diagnostic realism processing realistic modules of user code.

VII CONCLUSION

The work under this contract has shown by the examples of navigation and flight control that the CHAMP approach to architecture design can be applied to avionic computational problems. Furthermore, the simulations have shown that CHAMP can provide a potentially survivable computer for aircraft use.

CHAMP uses a simple network which can make use of advanced hardware techniques (such as those available through the VHSIC program). It also simplifies material inventory problems since there is only one type of spare (a single PC) which is contained in a single chip. The PC used in this study is well within the capability of single-chip techniques projected for DAIS the 1985 time period.

The avionics functions software used in this study were created by using DAIS project personnel structured programming techniques which yielded sub-modules for each function that were small and tractable and easily managed by microprocessor-based computers of modest capability. This is especially significant considering that this can be accomplished with processors operating at relatively low-frequency clock rates (1 MHz as opposed to the 10 to 20-MHz clock often used in more elaborate processors). This results in a much more reliable and easily constructed hardware module for the computer. Furthermore, this relative simplicity tends to make the computing modules immune to noise, interference, and EMP.

The communications algorithms designed under this contract have been shown to be effective managers of the network message traffic. In fact, traffic between the avionic modules characterized in this study is relatively modest. Even with the high overhead of broadcast and diagnostics this traffic probably will not provide an excessive load for relatively low-speed links between the processors. The higher-speed buses required in more elaborate processors can present a reliability problem, especially when field maintenance is involved.

For the case of signal processing, the amount of overhead relative to message traffic can be a significantly smaller fraction than in the case of non-signal processing problems. The most efficient communication method for signal processing is sending a percursor message to open the necessary communication links between processors so that the message can be sent in a burst at full link bandwidth.

Fault tolerance is a hierarchical design problem, and the degree of damage resistance that is needed can be designed. The CHAMP-style architecture provides a top-down approach to the creation of computer hardware for military applications. Using the CHAMP techniques as illustrated by this study, it is possible first to define the needs of the operational environment, the threat to be met, and the weight and power costs that are acceptable; and then define the type of network and operating system to solve the problem. The CHAMP approach, which uses a multiplicity of identical resources under control of a distributed operation system, provides a step-by-step process for achieving this result. Not all of the attendant engineering problems are solved, but the direction in which to proceed is clear.

The final point discussed in this report is that the simulations indicate that the CHAMP network has a capability for survival. Although the simulations do not show the detailed behavior of actual hardware, they do suggest that the system's survival characteristics are good. More work is needed in order to identify weak points and to demonstrate and develop specific hardware. An initial design for the load-levelling and message-traffic operating system is provided in this report in Appendices A and B. These appendicies can be used to write the code for a hardware simulation.

This study has shown the feasibility of network computing for avionics. However, investigation and demonstration still is needed in several areas, including: (1) extension of the study to include an analysis of computer code for all avioncis functions, (2) modification of the SRI CHAMP hardware network to allow for demonstration of the

survivability of avionics computing capabilities using real code modules, and (3) extension of the work on load control and communications algorithms.

Appendix A

PROGRAM DESIGN FOR NETWORK COMMUNICATION

CHAMP SYSTEM SIMULATION

Simulation Setup
Simulation Run
Simulation Results Evaluation

1. Simulation Setup

Determine characteristics of software blocks

(inputs, outputs, priorities, memory requirements, processing time requirements)

Allocate software blocks to processors

("LOCAL" is which blocks are run by each processor)
(Backup copies must also be distributed)

Establish system failure schedule

2. Simulation Run

Fail processors and links as scheduled

Manage Link Traffic

Pass messages between I/O processors
Garble messages as required
Lose messages as required

Manage processor functions

Damage Supervisor

Perform self check
Accept requests for checking
Run diagnostics
Send results to neighbor

Perform Neighborhood Patrol

Determine what neighbor to check
Perform request for status

Perform Load Management Function

Manage Task Processor

Start RU
Choose next RU for execution
Maintain active RUs
Pass data into/out of task processor

PROGRAM DESIGN STRUCTURE

Manage I/O Processor

Process Messages From Neighbors

Prepare local acknowledgement messages
Remove andprocess previously seen messages
Update route information
Sort messages for distribution/relay
Process timed out messages
Process unacknowledged task processor data output messages

Process Messages For Transmission From Supervisor Task Processor

Accept messages from supervisor processor
Prepare message for transmission

Send Messages to Neighbors

Perform route and node determination algorithms
Update Sent and Backup data structures

Manage Task Processor

Accept messages from supervisor
Run RU or diagnostics
Send data or results to supervisor

3. Simulation Evaluation

Monitor traffic load
Monitor accomplishment of scheduled tasks
Monitor system performance after failure

PROGRAM DESIGN STRUCTURE

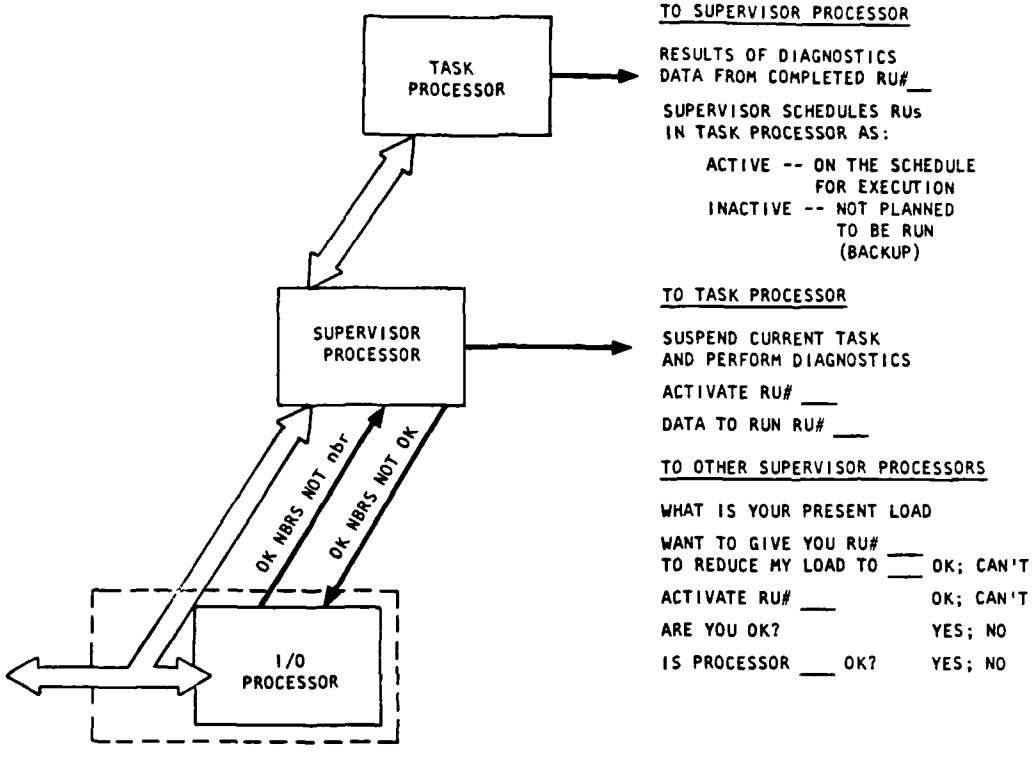


FIGURE A-1 PROCESSING CENTER CONTROL MESSAGE TRAFFIC

MODULE FUNCTIONAL DESIGN SPECIFICATION I

MODULE NAME: Manage I/O Processor Rev No: 3

AUTHOR: C. Monson Date: 17 July 1980

ONE-LINE DESCRIPTION: Processes message traffic into and out of processing center and messages originating within task or supervisor processor.

DETAILED DESCRIPTION: (Each time this module is called, it sequentially calls each submodule exactly once).

Process messages from neighbors (I.A)

- Prepare local acknowledgement messages
- Remove and process previously seen messages
- Update route information
- Sort messages for distribution/relay
- Process unacknowledged local transmissions
- Process unacknowledged task processor data output messages

Process messages for transmission (I.B)

- Accept messages from supervisor processor
- Prepare messages for transmission

Send messages to neighbors (I.C)

- Perform route and mode determination algorithm
- Update SENT and BACKUP data structures

INPUTS: Incoming messages from other I/O processors; messages from task and supervisor processors; control messages from supervisor processors.

OUTPUTS: Outgoing messages from this I/O processor, supervisor and task processor; messages to supervisor processors.

THIS MODULE CALLED BY: Manage Processor Functions

THIS MODULE CALLS: Process messages from neighbors; process messages for transmission; and send messages to neighbors.

UNDEFINED ASPECTS:

RU data output:

# Data Words	Msg. Mode	Originating RU	Destination RU	Msg. Type
6 bits	2 bits	8 bits	8 bits	$FF = orig. msg.

RU data input:

# Data Words	Msg. Mode	Originating RU	Destination RU
6 bits	2 bits	8 bits	8 bits

Control msg: (neighbor diagnostics, load level, etc.)

Control Msg.	Msg. Mode	Destination PC	Msg. Type
6 bits	2 bits	8 bits	$FF = orig. msg.

FIGURE A-2 SUPERVISOR PROCESSOR/I/O PROCESSOR MESSAGE INTERCHANGE

RU data input/output:

# Data Words	Msg. Mode	Originating RU	Destination RU	Msg. Type	Message ID: Originating PC	Message ID: No. from ID Counter
6 bits	2 bits	8 bits	8 bits	$FF = orig. msg.	8 bits	8 bits

Acknowledgement message:

# Data Words	Msg. Mode	Originating RU	Destintion RU	Msg. Type	Message ID of Orig. Msg.	
6 bits	2 bits	8 bits	8 bits	$FØ = ack. msg.	8 bits	8 bits

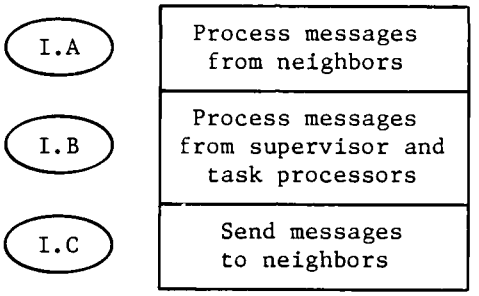
Local acknowledgement message:

Msg. Type	Message ID of Orig. Msg.	
$ØØ = local ack.	8 bits	8 bits

Control message--from supervisor to suprvisor:

Control Msg.	Msg. Mode	Destination PC	Msg. Type	Message ID: Originating PC	Message ID: No. from ID Counter
6 bits	2 bits	8 bits	$FF = orig. msg.	8 bits	8 bits

FIGURE A-3 I/O PROCESSOR TO I/O PROCESSOR MESSAGE INTERCHANGE



MANAGE I/O PROCESSOR

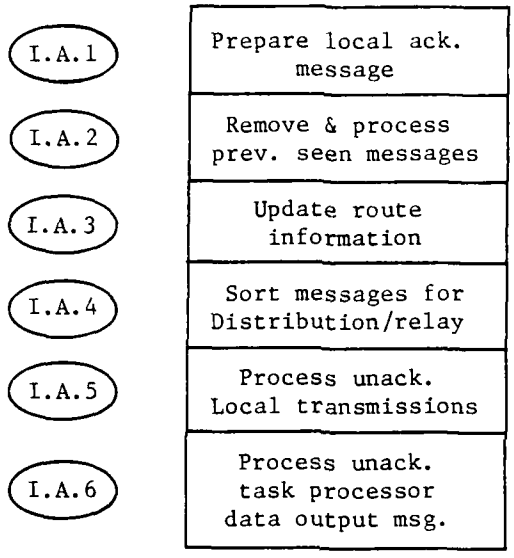
MODULE FUNCTIONAL DESIGN SPECIFICATION

MODULE NAME: Process Message from Neighbors Rev No: 0

AUTHOR: E. Wischmeyer Date: 24 June 1980

ONE-LINE DESCRIPTION: Processes all messages received from neighboring I/O processors and performs route determination algorithm.

DETAILED DESCRIPTION: (Sequentially calls five submodules; each one called exactly once whenever this module called).



INPUTS: All messages received from neighbors; route data structure for this I/O processor.

OUTPUTS: Messages to be eventually sent to other I/O processors; messages to this task and supervisor processors.

THIS MODULE CALLED BY: Manage I/O Processor

THIS MODULE CALLS:

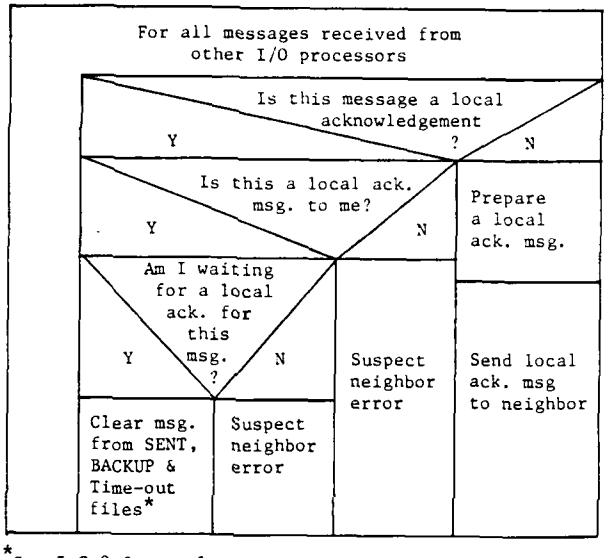
UNDEFINED ASPECTS:

MODULE NAME: Prepare Local Acknowledgement Message Rev No: 0

AUTHOR: C. Monson Date: 19 June 1980

ONE-LINE DESCRIPTION: Send local acknowledgement to neighbor from whom message received.

DETAILED DESCRIPTION:



*See I.C.2 for explanation and setup of SENT, BACKUP and TIMEOUT files

INPUTS: All received messages from other I/O processors; SENT, BACKUP and TIMEOUT files

OUTPUTS: Local acknowledgement messages to neighbors; received messages that are not local acknowledgement; SENT, BACKUP and TIMEOUT files

THIS MODULE CALLED BY: Process messages from neighbors

THIS MODULE CALLS:

UNDEFINED ASPECTS: What to do with suspected neighbor error in local acknowledgement.

MODULE FUNCTIONAL DESIGN SPECIFICATION I.A.2

MODULE NAME: Remove and Process Previously Seen Messages Rev No: 0

AUTHOR: C. Monson Date: 19 June 19

ONE-LINE DESCRIPTION: Messages are sorted and checked if messages which have been seen previously are abondoned; new messages continue.

DETAILED DESCRIPTION:

Sort out acknowledgement message (except local acknowledgement)

Remove this message from list of messages previously seen if original message seen before.

Store acknowledgement message in list of acknowledgement previously seen if acknowledgement not previously seen. Abandon acknowledgement message if acknowledgement previously seen.

Compare message with list of messages previously seen (non-acknowledgement message).

Abandon message if previously seen.
Store message if not previously seen.

(Note: Need only store message ID in data structures created above as record of messages previously seen).

INPUTS: Received messages that are not local acknowledgement

OUTPUTS: New messages to be shipped to supervisor, task, or other I/O processors

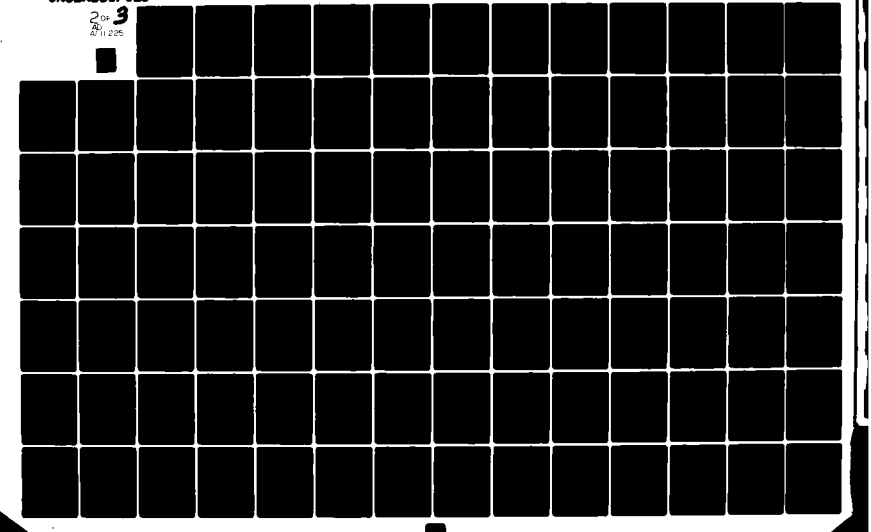
THIS MODULE CALLED BY: Process Messages from Neighbors

THIS MODULE CALLS: Sort out acknowledgement message; compare message with list o messages previously seen

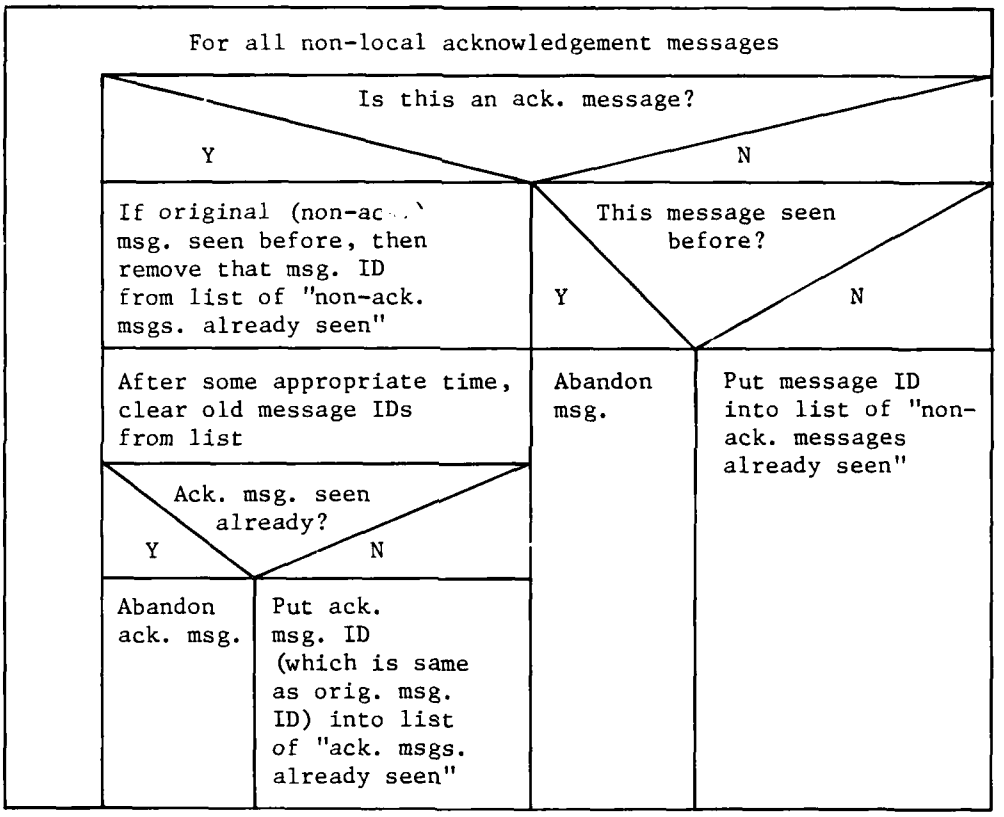
UNDEFINED ASPECTS:

AD-A111 225 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
SURVIVABLE AVIONICS COMPUTER SYSTEM.(U)
NOV 80 P R MONSON, C A MONSON, H C PEASE F33615-80-C-1014
UNCLASSIFIED NL

2 OF 3
AD
A111225



REMOVE AND PROCESS PREVIOUSLY SEEN MESSAGES



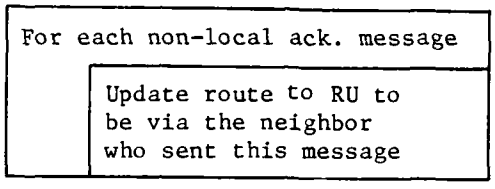
MODULE FUNCTIONAL DESIGN SPECIFICATION I.A.3

MODULE NAME: Update Route Information Rev No: 0

AUTHOR: C. Monson Date: 19 June 1980

ONE-LINE DESCRIPTION: Neighbor who sent message and name of originating RU or center are stored for local route information to that RU or center.

DETAILED DESCRIPTION:



INPUTS: Neighbor sending message; name of RU originating message

OUTPUTS: Route information data structure

THIS MODULE CALLED BY: Process Messages from Neighbors

THIS MODULE CALLS:

UNDEFINED ASPECTS: Design of route information data structure

MODULE FUNCTIONAL DESIGN SPECIFICATION I.A.4

MODULE NAME: Sort Messages for Distribution/Relay Rev No: 4

AUTHOR: C. Monson Date: 17 July 1980

ONE-LINE DESCRIPTION: Separate out messages for further relay and process local messages.

DETAILED DESCRIPTION:

Send Broadcast messages to supervisor processor

Separate out messages for further relay

Process messages for this processing center.
Give other messages to supervisor processor.
Give messages to supervisor for task processor (data messages).
Relay copy of message so that all identical RUs in network (active and incative) receive updates.
Send acknowledgement if message is for active RU.

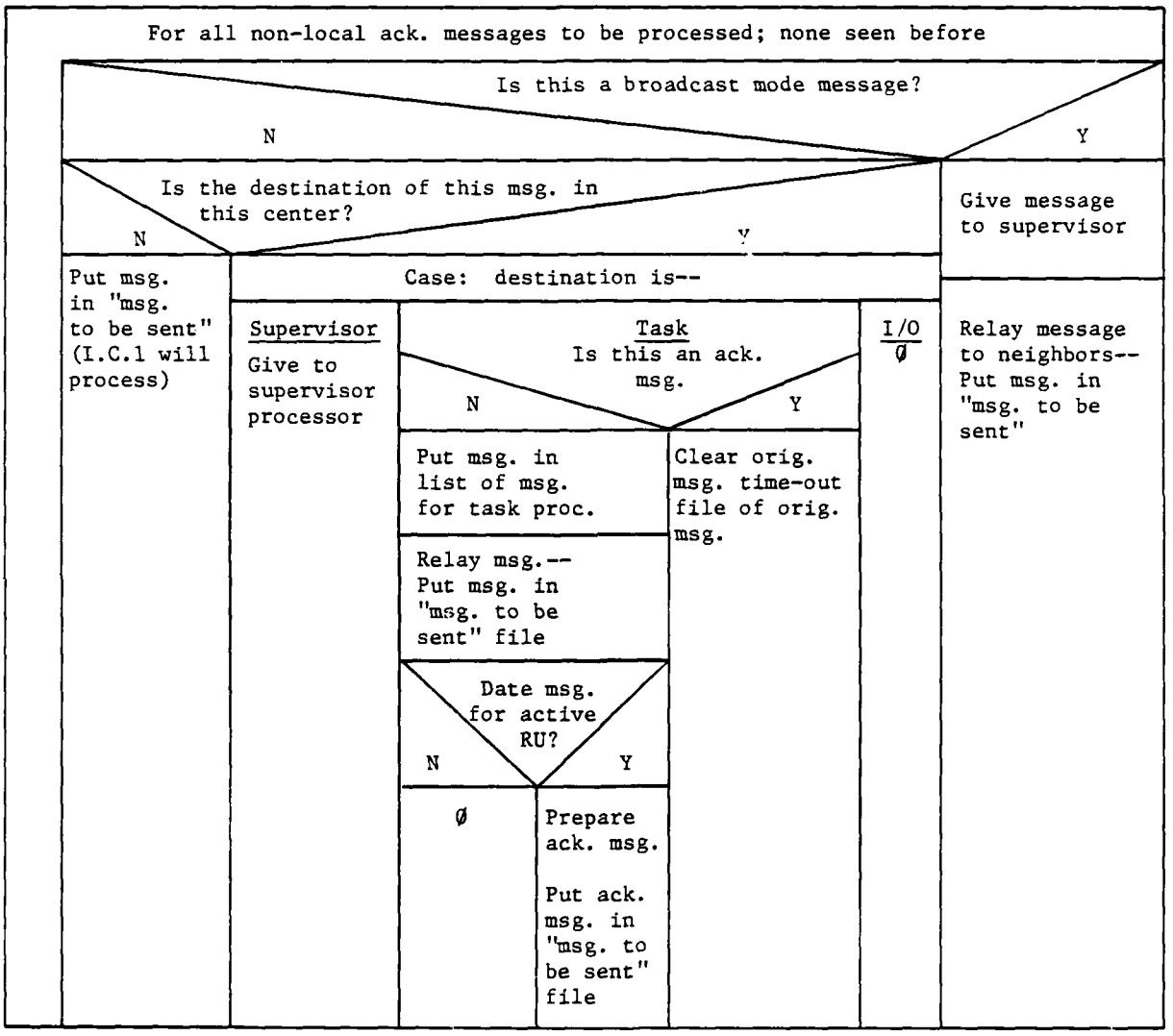
INPUTS: Messages received from other I/O processors that are not local acknowledgement messages, list of all RUs (active and inactive) in this center.
OUTPUTS: Task and supervisor processor messages; messages for relay to neighbors, ORIG MISG TIMEOUT

THIS MODULE CALLED BY: Process Messages from Neighbors

THIS MODULE CALLS: Send broadcast messages to supervisor processor; separate out messages for further relay; process messages for this processing center.

UNDEFINED ASPECTS:

SORT MESSAGES FOR DISTRIBUTION/RELAY



MODULE FUNCTIONAL DESIGN SPECIFICATION I.A.5

MODULE NAME: Process Unacknowledged Local Transmissions Rev No: 0

AUTHOR: E. Wischmeyer Date: 26 June 1980

ONE-LINE DESCRIPTION: When messages have timed-out without a local acknowledgement this module takes corrective steps.

DETAILED DESCRIPTION:

After all of the incoming local acknowledgements have been processed, this module examines those messages for which a local acknowledgement has not been received. If these messages have "timed-out", the following steps are taken:

1. The processor not sending local acknowledgement is removed from OK Neighbors list.

2. The I/O processor informs the supervisor of this action.

3. The unacknowledged message is resent to other processors, if required. If the message was sent in search, broadcast, or default modes, there is no need to resend the message-because all OK neighbors have already seen the message. If the message was sent in directed mode, it can be resent in default mode to all OK neighbors. The processor from whom the message was originally received will ignore the message.

If the message has not yet "timed-out", increment the counter.

INPUTS: Message awaiting local acknowledgement (sent and backup lists); OK neighbors list.

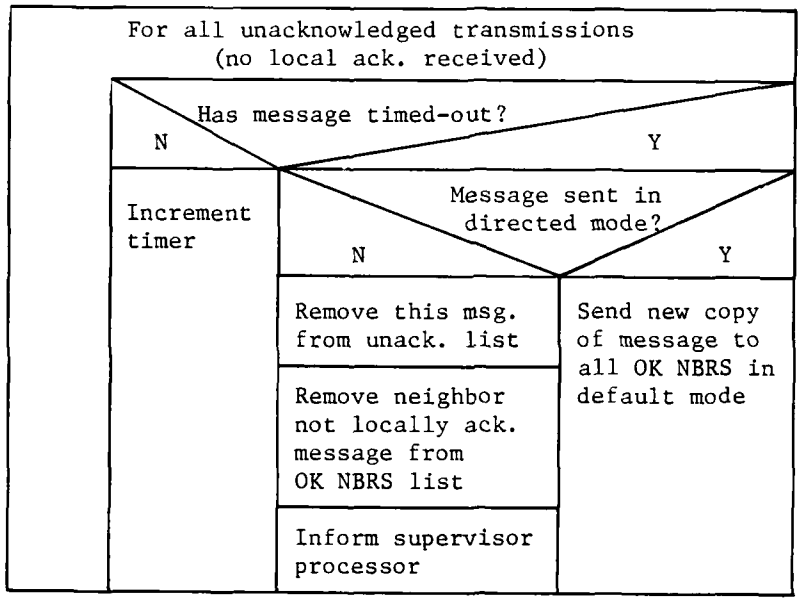
OUTPUTS: Message awaiting local acknowledgement; OK neighbors list; message to supervisor processor.

THIS MODULE CALLED BY: Process Messages from Neighbors

THIS MODULE CALLS:

UNDEFINED ASPECTS: Time dealy for timed-out messages

PROCESS UNACKNOWLEDGED LOCAL TRANSMISSIONS



MODULE FUNCTIONAL DESIGN SPECIFICATION I.A.6

MODULE NAME: Process Unacknowledged Task Processor Data Output Message **Rev No:** 0

AUTHOR: C. Monson **Date:** 11 July 1980

ONE-LINE DESCRIPTION: When data output messages from task processor do not receive an acknowledgement from the destination PC, corrective action takes place.

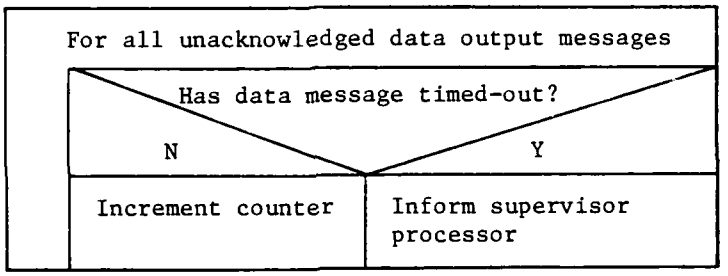
DETAILED DESCRIPTION:

Examine original message timeout for messages with overdue acknowledgement message

If message has "timed-out":

Notify supervisor processor (who in turn sends message to backup RU to become active).

If message not "timed-out," increment counter



INPUTS: Original message time-out data structure

OUTPUTS: Original message timeout data structure; message to supervisor processor

THIS MODULE CALLED BY: Process Messages from Neighbors

THIS MODULE CALLS:

UNDEFINED ASPECTS: Time delay for timed-out data output messages from task processor

MODULE FUNCTIONAL DESIGN SPECIFICATION I.B

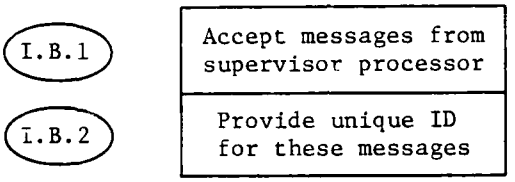
MODULE NAME: Process Messages for Transmission Rev No: 1

AUTHOR: E. Wischmeyer Date: 17 July 1980

ONE-LINE DESCRIPTION: Accepts messages for transmission from supervisor processor.

DETAILED DESCRIPTION:

(Each time this module is called, it sequentially calls each submodule exactly once).



INPUTS: Messages originated by supervisor and task processors

OUTPUTS: Messages to be sent to neighbors

THIS MODULE CALLED BY: Manage I/O Processor

THIS MODULE CALLS:

UNDEFINED ASPECTS:

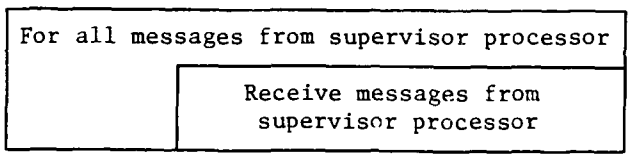
MODULE FUNCTIONAL DESIGN SPECIFICATION I.B.1

MODULE NAME: Accept Messages From Supervisor Processor Rev No: 0

AUTHOR: C. Monson Date: 19 July 1980

ONE-LINE DESCRIPTION: Accepts messages from supervisor processor for transmission to neighbors

DETAILED DESCRIPTION:



INPUTS: Messages for transmission from supervisor processor

OUTPUTS: Originating messages, not local acknowledgement

THIS MODULE CALLED BY: Process Messages for Transmission from Supervisor and Task Processors

THIS MODULE CALLS:

UNDEFINED ASPECTS:

MODULE FUNCTIONAL DESIGN SPECIFICATION I.B.2

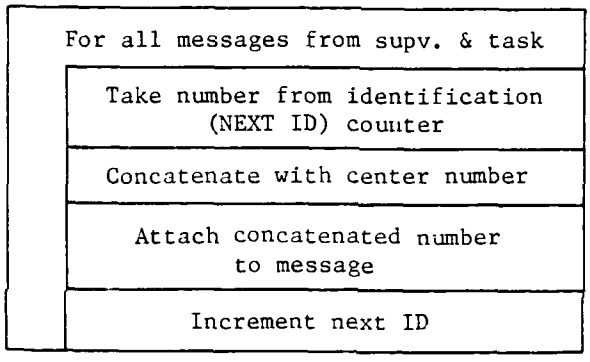
MODULE NAME: Prepare Messages for Transmission Rev No: 0

AUTHOR: C. Monson Date: 17 June 1980

ONE-LINE DESCRIPTION: Pulls number from counter and concatenates with center number to form unique identification for message.

DETAILED DESCRIPTION:

Messages are prepared for transmission by providing an identification number that is unique to that message and may be stored by centers seeing the message to keep track of messages already seen and message acknowledgements. The ID is formed by concatenating the number of the processing center and a number generated by a counter. The counter will be incremented every time a new ID is created. The counter may be reset after a time when old messages are surely no longer in use.



INPUTS: Messages from task or supervisor processor; center number; NEXT ID number.

OUTPUTS: Uniquely identified message for transmission; new NEXT ID number

THIS MODULE CALLED BY: Process Messages for Transmission from Supervisor and Task Processors

THIS MODULE CALLS:

UNDEFINED ASPECTS:

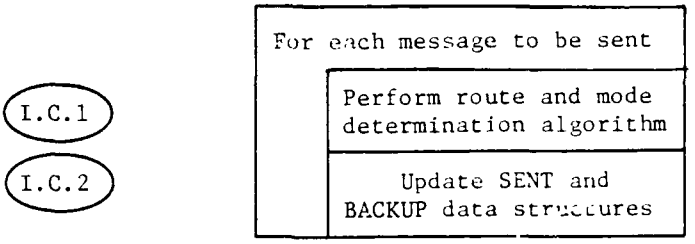
MODULE FUNCTIONAL DESIGN SPECIFICATION I.C

MODULE NAME: Send Messages to Neighbors Rev No: 0

AUTHOR: C. Monson Date: 27 June 1980

ONE-LINE DESCRIPTION: Determines route for sending message and mode to be sent and stores message until successfully transmitted.

DETAILED DESCRIPTION:



INPUTS: Messages for transmission to neighbors

OUTPUTS: Messages to specified neighbor(s) in particular sending mode

THIS MODULE CALLED BY: Manage I/O Processor

THIS MODULE CALLS:

UNDEFINED ASPECTS:

MODULE FUNCTIONAL DESIGN SPECIFICATION I.C.1

MODULE NAME: Perform Route and Mode Determination Algorithm Rev No: 0

AUTHOR: C. Monson Date: 17 June 1980

ONE-LINE DESCRIPTION: Look for available route information, check if route working, identify message transmission mode.

DETAILED DESCRIPTION:

Obtain route information unless message received in DEFAULT mode.

Compare route with list of working neighbors (OK neighbor).

Set message sending mode unless message received in DEFAULT mode.
SEARCH mode set when no route information available.
DIRECTED mode set when route information available and route works (neighbor OK).
DEFAULT mode set when route information available but neighbor not working.

INPUTS: Message destination, route information, OK neighbor list; messages to be sent/relayed to neighbors

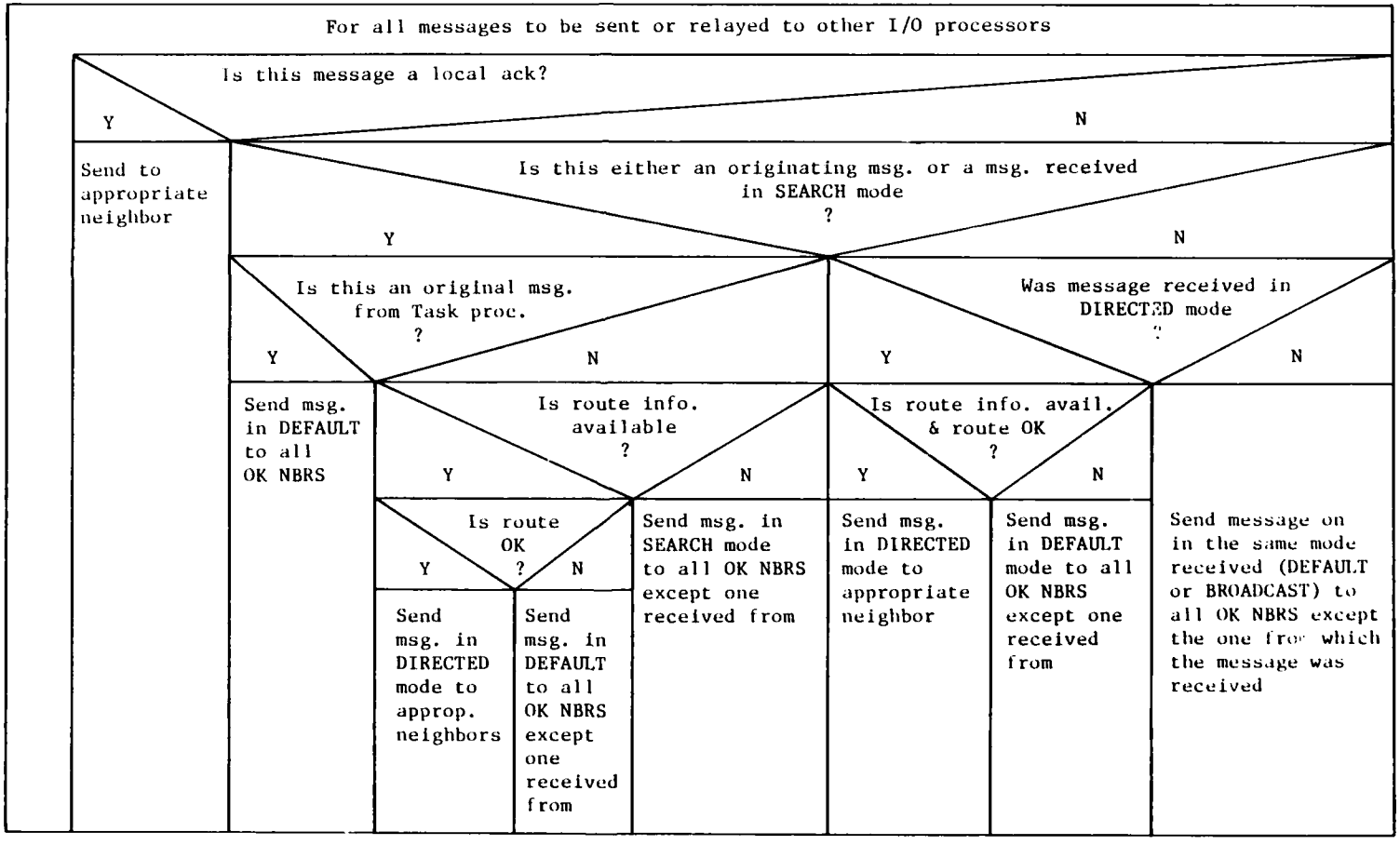
OUTPUTS: Message route (neighbor(s) to send message), message mode for each message

THIS MODULE CALLED BY: Send Messages to Neighbors

THIS MODULE CALLS: Obtain route information; compare route with list of working neighbors; and set message sending mode

UNDEFINED ASPECTS:

PERFORM ROUTE AND MODE DETERMINATION ALGORITHM



MODULE FUNCTIONAL DESIGN SPECIFICATION I.C.2

MODULE NAME: Update Sent and Backup Data Structures **Rev No:** 2

AUTHOR: C. Monson **Date:** 21 July 1980

ONE-LINE DESCRIPTION: Enter messages and message data into files and wait for successful transmission.

DETAILED DESCRIPTION:

Set timer for message locak acknowledgement TIMEOUT.

If originating data output message from task processor, store ID and departure time in original message TIMEOUT file.

If message mode is directed:
Store message and ID in Backup file.
Store message ID and neighbor sent to in Sent file.

(Sent, Backup and TIMEOUT files are set up so that if, after some reasonable time, the transmission of a message is not successful because of neighbor or link failure, failed neighbors can be reported to the supervisor processor and if the message was sent in directed mode, it can be pulled from the Backup file and be re-transmitted).

INPUTS: All non-local acknowledgement messages to be sent to other I/O processors

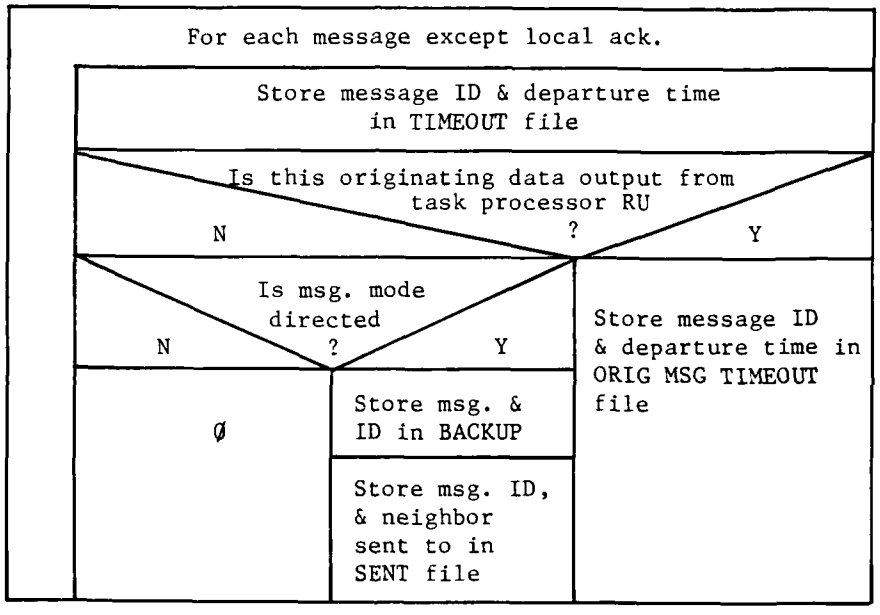
OUTPUTS: New entries in Backup and Sent files, and set timers for message transmission success

THIS MODULE CALLED BY: Send Messages to Neighbors

THIS MODULE CALLS:

UNDEFINED ASPECTS:

UPDATE SENT AND BACKUP DATA STRUCTURES



	I.A					I.B		I.C	
	I.A.1	I.A.2	I.A.3	I.A.4	I.A.5	I.A.6	I.B.1	I.B.2	I.A.1	I.A.2
All received msgs. from other I/O proc.	R									
All outgoing msgs. to other I/O proc.									W	R
Routing data			U							
Msg. to this supv. processor				W	W	W				
Msg. from this supv. processor							R			
SENT message	U				U					U
BACKUP message	U				U					U
Local ack. time-out	U				U					U
ORIG MSG time-out				U		U				U
Non-ack. msgs. already seen		U								
Ack. msgs. already seen		U								
Local ack. msgs. to neighbors	W									
Msgs to be sent/relayed				W	W					R
Received msgs. not local ack.	W	R	R	R						
Originating msgs. not local ack.								W U		R
OK NBRS*					W					R
RUs local to this task processor*				R						

*File common to I/O processor & supervisor processor

R = reads from file
W = writes to file
U = updates file (R & W)

I/O PROCESSOR FILE STRUCTURE

Appendix B

PROGRAM DESIGN FOR DISTRIBUTED NETWORK EXECUTIVE

MODULE FUNCTIONAL DESIGN SPECIFICATION II

MODULE NAME: Manage Supervisor **Rev No:**

AUTHOR: A. Takahashi **Date:** 16 July 1980

ONE-LINE DESCRIPTION: Performs the following executive functions: error checking, load levelling, task processor scheduling.

DETAILED DESCRIPTION: (Modules are called in sequential order).

MANAGE SUPERVISOR

Perform self-check to determine status of this PC's task processor: IIA
Accept requests to do self check from other neighbors.

Tell task processor to run a diagnostics routine.

Send task processors to status to neighbors that requested it.

Perform neighbor patrol to revise status of this PC's connected neighbors: IIB
Determine what specific neighbors need to be checked.

Obtain neighbor status by requesting neighbor to do self-check and change good neighbor's list if neighbor is bad.

Perform Load Management: IIC
Distribute active RUs as evenly among local task processors as possible.

MANAGE TASK PROCESSOR EXECUTIVE IID

Accept and process requests to start up a RU.

Determine what RU to run next and request it to start execution.

Make sure there are active copies available of RUs to be run.

Pass data into/out of task processor.

INPUTS: Requests from I/O executive for self-check, OK neighbors list, status requests, order of RUs to be run.

OUTPUTS: Messages to I/O executive of status, requests for neighbor self-check, corrections to OK neighbors

THIS MODULE CALLED BY: Manage Processor Functions

THIS MODULE CALLS: Perform self-check; neighborhood patrol; perform load management

UNDEFINED ASPECTS:

MODULE FUNCTIONAL DESIGN SPECIFICATION IIA

MODULE NAME: Perform Self-Check **Rev No:**

AUTHOR: A. Takahashi **Date:** 17 July 1980

ONE-LINE DESCRIPTION: At the request of a neighbor, this routine requests the task processor to run a diagnostics program.

DETAILED DESCRIPTION:

Perform self-check to determine status of this PC's task processor.

Accept requests to do self-check from other neighbors.

Tell task processor to run diagnostics routine, and then determine task processor's status.

Send task processor status to neighbors that requested it.

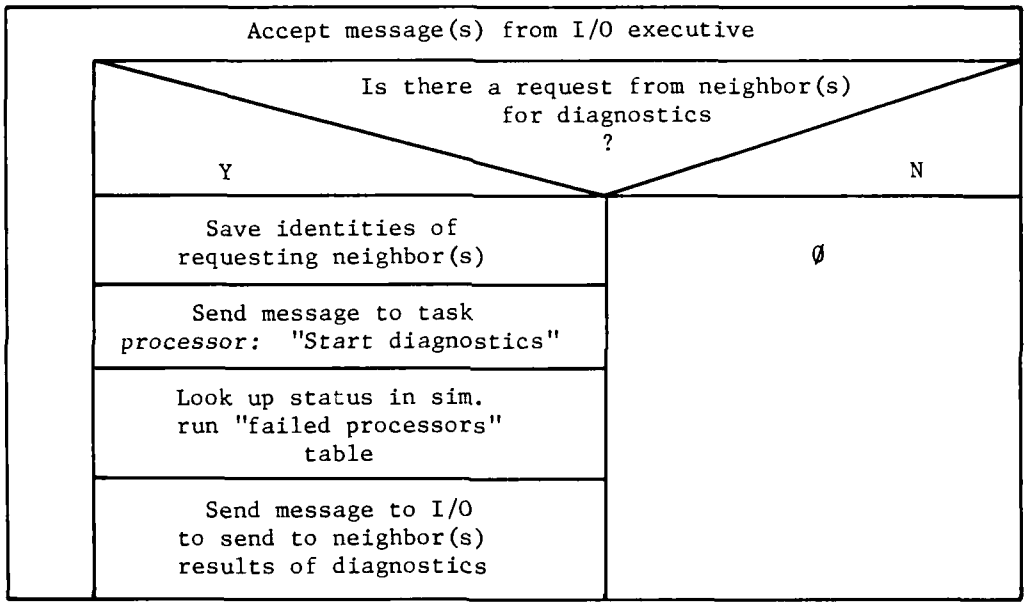
INPUTS: Messages from I/O executive (to do self-check, and ID of requesting PC), messages from task processor (have completed diagnostic routine), timer for diagnostics.
OUTPUTS: Message for I/O executive (status of task processor, list of neighbors to send status to), messages for task processor (start diagnostic routine).

THIS MODULE CALLED BY: Manage Supervisor

THIS MODULE CALLS:

UNDEFINED ASPECTS:

PERFORM DIAGNOSTICS TO DETERMINE STATUS OF THIS PC's TASK PROCESSOR



MODULE FUNCTIONAL DESIGN SPECIFICATION IIB

MODULE NAME: Perform Neighbor Patrol **Rev No:**

AUTHOR: A. TAkahashi **Date:** 17 July 1980

ONE-LINE DESCRIPTION: Determines status of PCs connected to this PC by asking a neighbor to perform self-check, verifies status by asking other neighbors to also check questionable PC and revises good neighbor's list (modules are run in sequential order).

DETAILED DESCRIPTION:

Perform neighbor patrol.

Accept/generate requests to do check on specific neighbor(s). IIB1

Obtain neighbor status by requesting neighbor to do self-check, and change good neighbor's list if neighbor is bad. IIB2

INPUTS: Requests for a check of specific neighbor(s), message from other PCs regarding status of questionable PC

OUTPUTS: Requests for another PC to do a specific neighbor check, revised good neighbor's list

THIS MODULE CALLED BY: Manage Supervisor

THIS MODULE CALLS: Accept/generate requests to do check; obtain neighbor status

UNDEFINED ASPECTS:

MODULE FUNCTIONAL DESIGN SPECIFICATION IIB1

MODULE NAME: Determine What Specific Neighbors to do Checks On **Rev No:**

AUTHOR: A. Takahashi **Date:** 17 July 1980

ONE-LINE DESCRIPTION: Will generate a list of neighbors to be checked--accepts outside requests for checks on a particular neighbor, otherwise returns the list of good neighbors.

DETAILED DESCRIPTION:

Determine what specific neighbors to do checks on:

Accept requests to check neighbors.
Obtain list of good neighbors.
Return a list of neighbors to be checked.

INPUTS: Messages from I/O executive (good neighbors' list, request to check this PC, ID of originating PC).

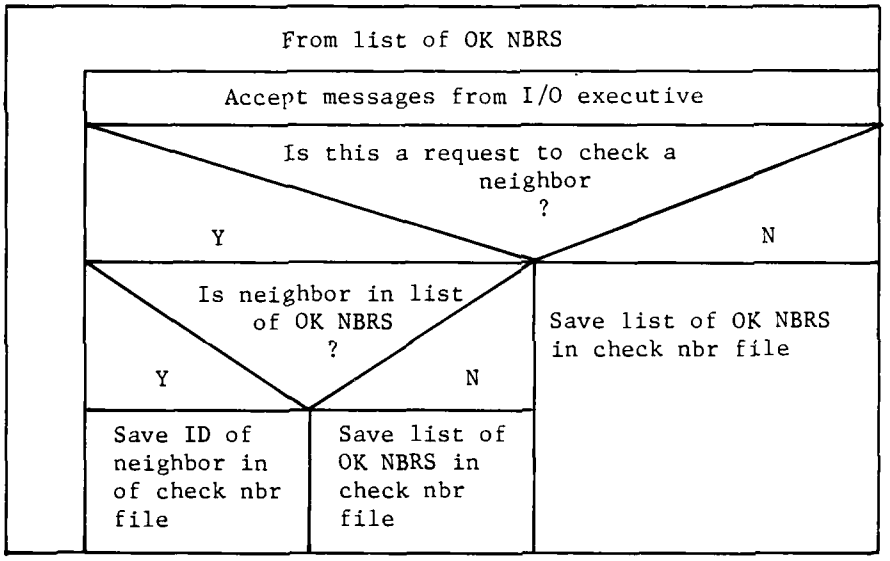
OUTPUTS: Check neighbors

THIS MODULE CALLED BY: Perform Neighbor Patrol

THIS MODULE CALLS:

UNDEFINED ASPECTS:

DETERMINE SPECIFIC NEIGHBOR TO BE CHECKED



MODULE FUNCTIONAL DESIGN SPECIFICATION IIB2

MODULE NAME: Obtain Neighbor Status **Rev No:**

AUTHOR: A. Takahashi **Date:** 17 July 1980

ONE-LINE DESCRIPTION: Send message to determine neighbor status and return results. If status of neighbor is bad, send a message to other neighbors to also start checking.

DETAILED DESCRIPTION:

Obtain neighbor status by requesting neighbor to do self-check:

Send messages to selected neighbors to do self-check.

Receive messages from neighbors regarding their task processor status.

If necessary, request other neighbors to begin neighbor check.

INPUTS: List of neighbors to check messages from neighbors (status of task processor) timing mechanism

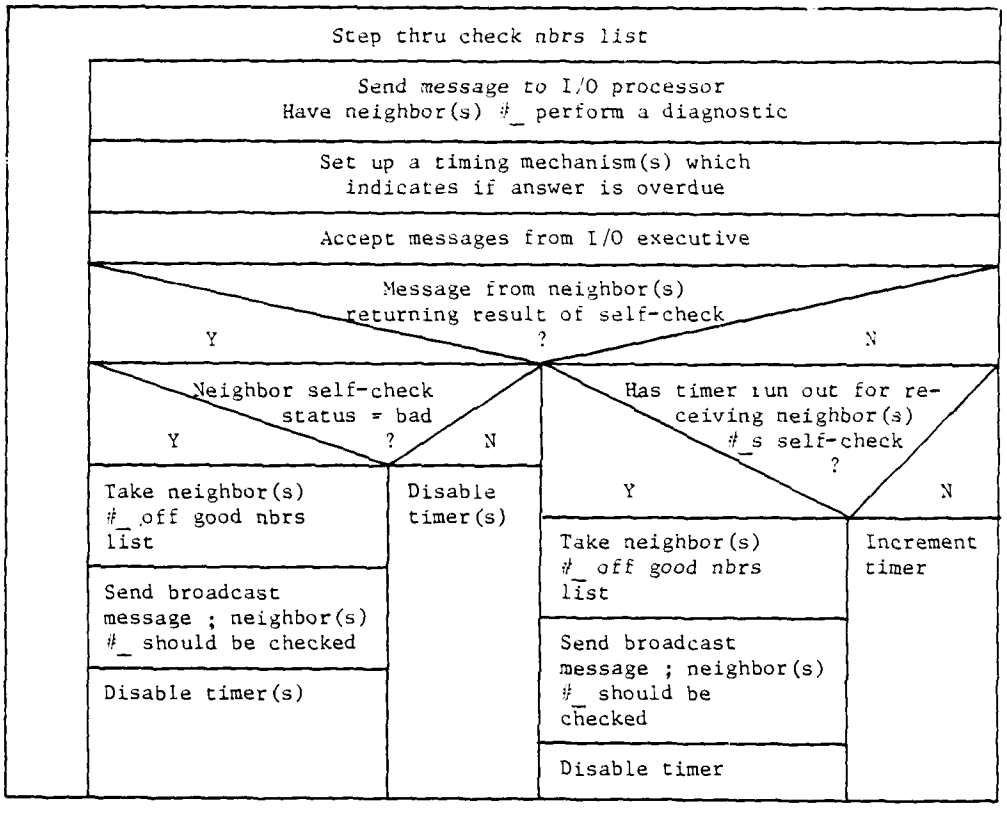
OUTPUTS: List of neighbors whose status is bad, messages to neighbor (do self-check and return results)

THIS MODULE CALLED BY: Perform Neighbor Patrol

THIS MODULE CALLS:

UNDEFINED ASPECTS:

OBTAIN NEIGHBOR STATUS BY REQUESTING NEIGHBOR TO DO SELF-CHECK



MODULE FUNCTIONAL DESIGN SPECIFICATION IIC

MODULE NAME: Perform RU Status and Load Management **Rev No:** 3

AUTHOR: A. Takahashi **Date:** 16 July 1980

ONE-LINE DESCRIPTION: Makes sure that processing is shared among neighbor PCs.

DETAILED DESCRIPTION: (Perform modules in sequential order).

Distribute active RU's as evenly as possible:

Select a neighbor to swap with on basis of which neighbor is most likely to benefit from swapping.	IIC1
Do a load evaluation for each RU swapped with selected neighbors and select the best RU to swap.	IIC2
Swap the proper RUs.	IIC3

INPUTS: List of good neighbors

OUTPUTS:

THIS MODULE CALLED BY: Manage Supervisor

THIS MODULE CALLS: Select neighbor, load evaluation, swap RUs

UNDEFINED ASPECTS:

MODULE FUNCTIONAL DESIGN SPECIFICATION IIC1

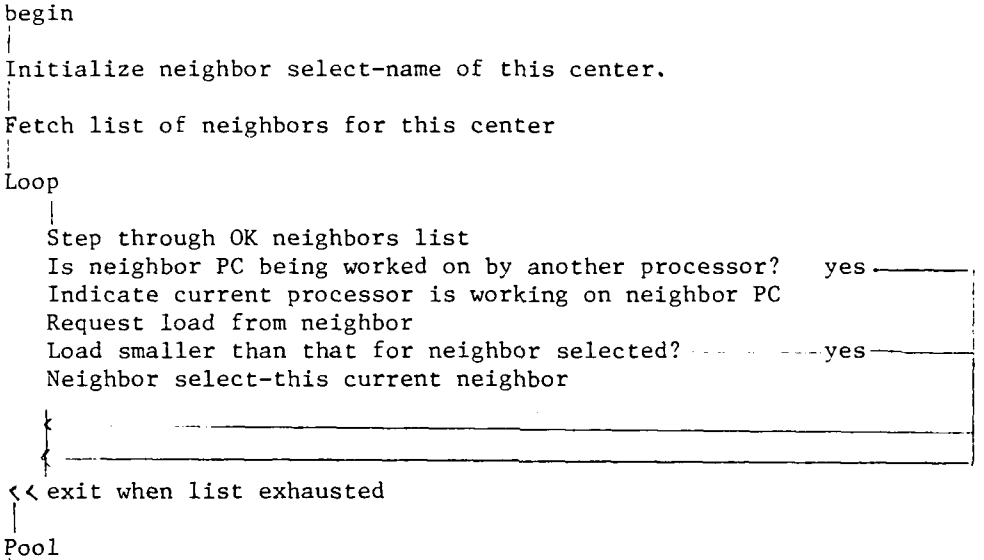
MODULE NAME: Select a Neighbor for Swapping **Rev No:** 2.3

AUTHOR: A. Takahashi **Date:** 14 July 1980

ONE-LINE DESCRIPTION: Determines which good neighbor has the smallest load currently, and is thus most likely to benefit from load levelling.

DETAILED DESCRIPTION:

Select a neighbor.



INPUTS: OK neighbors list, load-level busy flag, message from neighbor PC specifying its load

OUTPUTS: Neighbor select, request to neighbor PC for load information

THIS MODULE CALLED BY: Perform Load Management Function

THIS MODULE CALLS:

UNDEFINED ASPECTS: Need to time-out load-level busy flag

MODULE FUNCTIONAL DESIGN SPECIFICATION IIC2

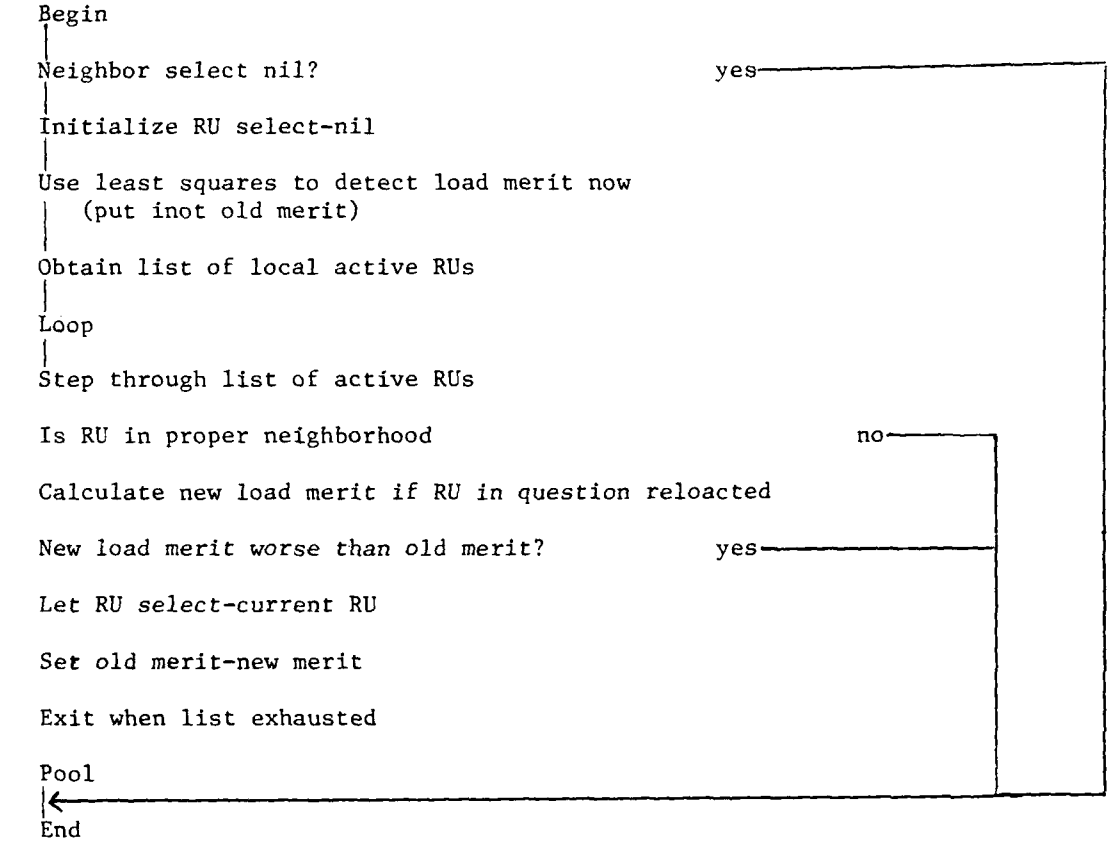
MODULE NAME: Load Evaluation **Rev No:** 2

AUTHOR: A. Takahashi **Date:** 16 July 1980

ONE-LINE DESCRIPTION: Using a least-square criterion, determines what RU is most advantageous to swap.

DETAILED DESCRIPTION:

Load Evaluation:



INPUTS: Neighbor select, list of local active RU's old-load merit, new-load merit.

OUTPUTS: RU select (name of RU to be swapped)

THIS MODULE CALLED BY: Perform Load Management Function

THIS MODULE CALLS:

UNDEFINED ASPECTS:

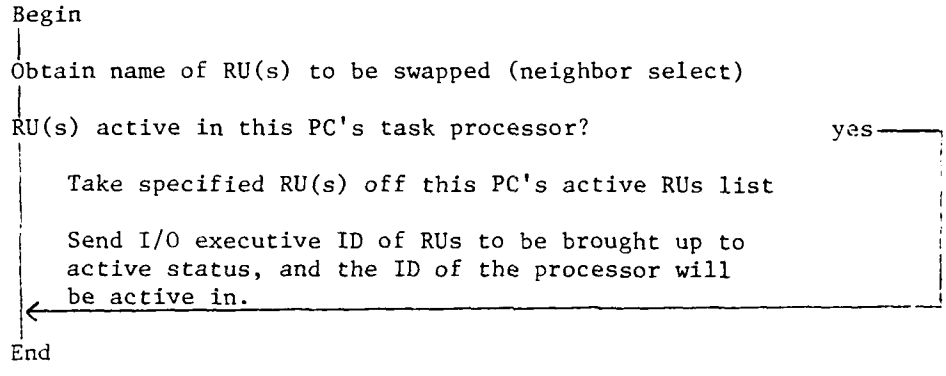
MODULE FUNCTIONAL DESIGN SPECIFICATION IIC3

MODULE NAME: Swap the Proper RUs **Rev No:**

AUTHOR: A. Takahashi **Date:** 16 July 1980

ONE-LINE DESCRIPTION: Gives messages to I/O executive specifying RUs which are to be swapped.

DETAILED DESCRIPTION:



INPUTS: Name of RU to be swapped (neighbor select), list of active RUs

OUTPUTS: Messages to other PC's, modified list of active RUs

THIS MODULE CALLED BY: Perform Load Management Function

THIS MODULE CALLS:

UNDEFINED ASPECTS:

MODULE FUNCTIONAL DESIGN SPECIFICATION IID

MODULE NAME: Manage Task Processor Executive **Rev No:**

AUTHOR: A. Takahashi **Date:** 16 July 1980

ONE-LINE DESCRIPTION: Determines which RUs are to be executed, makes sure an active copy is available or will be made available and gives the start RU commands.

DETAILED DESCRIPTION: (run in sequential order)

Manage task processor executive

Determine if data available to start up RU(s) (IID1)

Make sure there are active copies of RUs available to be run (IID2)

Pass data in/out of the task processor (IID3)

Determine which RU to run next and request it to start execution (IID4)

INPUTS: Messages from other PC's, messages from task processor executive

OUTPUTS: Messages to task processor executive

THIS MODULE CALLED BY: Manage Supervisor

THIS MODULE CALLS:

UNDEFINED ASPECTS:

MODULE FUNCTIONAL DESIGN SPECIFICATION IID1

MODULE NAME: Determine When to Begin RU(s) **Rev No:**

AUTHOR: A. Takahashi **Date:** 22 July 1980

ONE-LINE DESCRIPTION: Checks to see if data is available to run an RU. Also starts timers which are used for determining hwen an RU should have been activated; this being an indicator of a probable active RU loss or failure.

DETAILED DESCRIPTION:

Determine if data available to start up an RU(s).

Check data structures to determine what data has been received.

Put ID of those RUs with enough data to run on a list of RUs to be run.

INPUTS: List of data received by RUs, active RU's list, timer of active RU usage.

OUTPUTS: Run RU next list, modified timers, modified data list

THIS MODULE CALLED BY: Manage Task Processor Executive

THIS MODULE CALLS:

UNDEFINED ASPECTS:

MODULE FUNCTIONAL DESIGN SPECIFICATION IID2

MODULE NAME: Are Active Copies of RUs Available to be Run **Rev No:**

AUTHOR: A. Takahashi **Date:** 22 July 1980

ONE-LINE DESCRIPTION: If no acknowledgements are received for data transmission, or if an active RU has not been called within a reasonable amount of time, bring the back-up of the RU presummed bad to active status in the PC and prepare to start it up.

DETAILED DESCRIPTION:

Make sure there are active copies of RUs available to be run.

Check for non-acknowledgement of data.

Check timers on active RUs to see if RU should have been called.

Activate back-up RUs.

INPUTS: Messages from I/O executive, timing mechanism, li-t of active RUs.

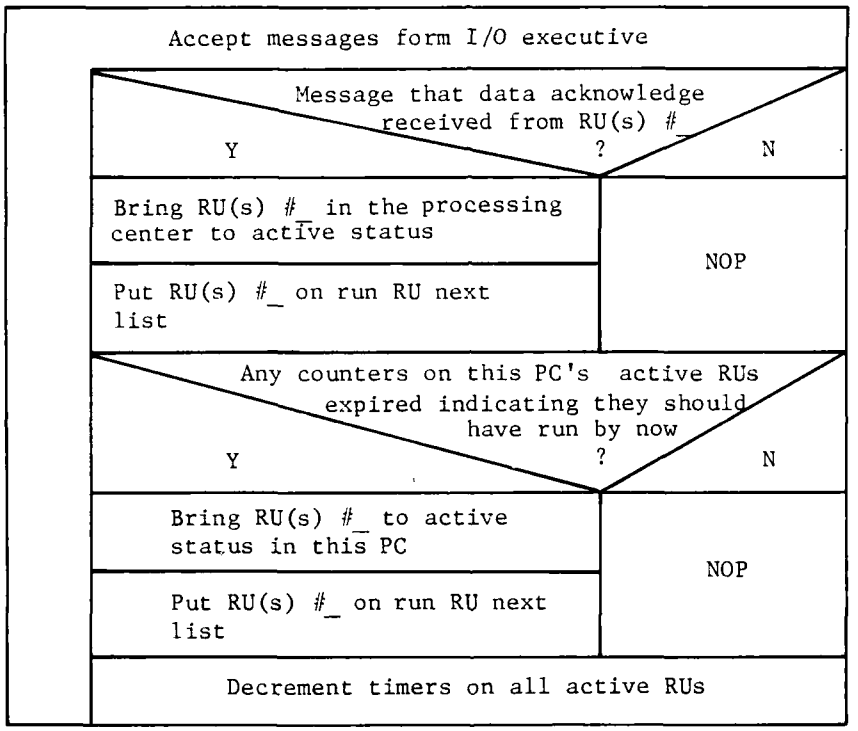
OUTPUTS: Run RU next list, list of active RUs,

THIS MODULE CALLED BY: Manage Task Processor Executive

THIS MODULE CALLS:

UNDEFINED ASPECTS:

ARE ACTIVE COPIES OF RUs AVAILABLE TO BE RUN?



MODULE FUNCTIONAL DESIGN SPECIFICATION IID3

MODULE NAME: Pass Data In/Out of Task Processor **Rev No:**

AUTHOR: A. Takahashi **Date:** 22 July 1980

ONE-LINE DESCRIPTION: Supplies data to, and accepts data from, RUs active in the task processor. Also keeps track of the RUs which are running.

DETAILED DESCRIPTION:

Pass data in/out of task processor.

Accept data from I/O executive or task processor.

Transmit data to task processor or I/O executive.

Take RUs which have finished running off of running RUs list.

INPUTS: Messages from I/O executive (RU data), messages from task processor (RU data), running RUs list, timing mechanism, active RUs list

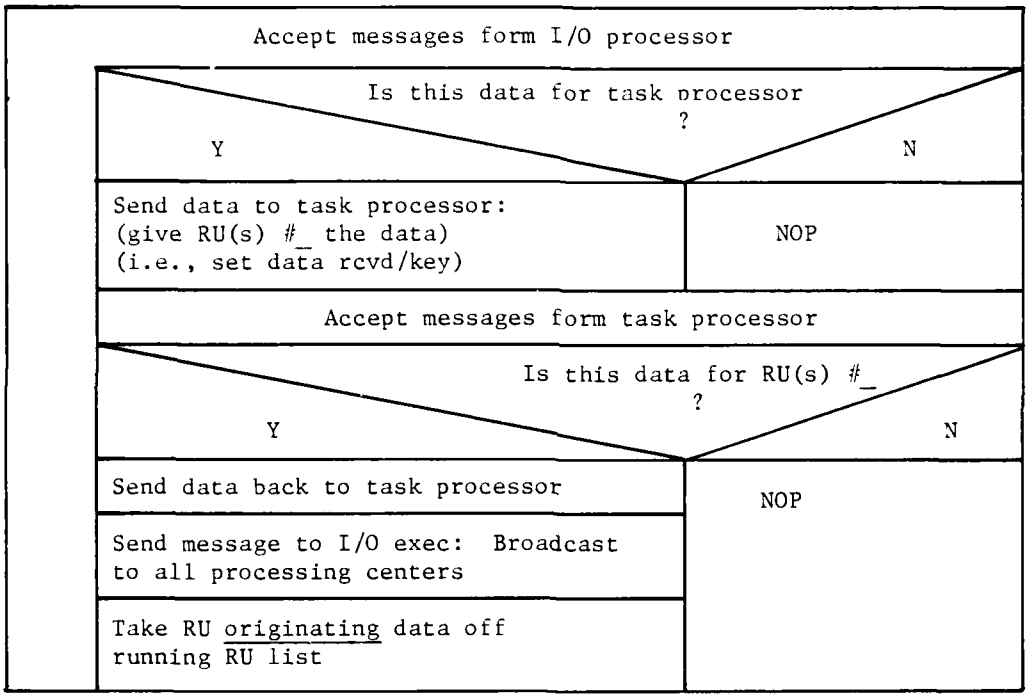
OUTPUTS: Messages to I/O executive (RU data), messages to task processor (RU data), modified running RUs list, modified timing mechanism

THIS MODULE CALLED BY: Task Processor Executive

THIS MODULE CALLS:

UNDEFINED ASPECTS:

PASS DATA IN/OUT OF TASK PROCESSOR



MODULE FUNCTIONAL DESIGN SPECIFICATION IID4

MODULE NAME: Determine Which RU to Run Next and Request it to Start Execution **Rev No:**

AUTHOR: A. Takahashi **Date:** 22 July 1980

ONE-LINE DESCRIPTION: Since the number of RUs required to run may exceed task processor's capacity, this module decides which will run first.

DETAILED DESCRIPTION:

Determine which RU to run next and request it to start execution.

Sort all Rus to be run by priority.

Choose a subset of all Rus to be running which will fit task processor's capability.

Run/suspend RUs as necessary to implement the above.

NOTE: Running RUs are those RUs which are currently running, run RU next are RUs which are executed next.

INPUTS: Running RU list, run RU next, data structure containing RU priorities, task processor capacity

OUTPUTS: Messages to task processor, modified RU next list, modified running RU's list (suspend/run RU)

THIS MODULE CALLED BY: Task Processor Executive

THIS MODULE CALLS:

UNDEFINED ASPECTS:

DETERMINE WHICH RU TO RUN NEXT AND REQUEST IT TO START EXECUTION

Temporarily combining running RUs and run RU next lists

Task processor capacity large enough to run everything ? Y	N
Send message to task processor to start all RU(s) in run RU next list. (add to running RU's list)	Sort combined list by RU priority
	Starting from highest priority RU, compile a list of RUs which will fit in the task processor
Delete all entries in run RU next list	Send message to task processor: Suspend RU(s) #_ (those RUs which are currently running, but not on the list compiled above)
	Send message to task processor: Start RU(s) #_ (those RUs in run RU next which appear in the above list)
	Delete RU(s) which were just started from run RU next list

	II.A	II.B.1	II.B.2	II.C.1	II.C.2	II.C.3	II.D.1	II.D.2	II.D.3	II.D.4
Task processor status table(s)*	R									
IDs of requesting neighbor	W									
Check neighbors		W	R							
Diagnostics timer			U							
Neighbor select (for load level)				U	R	R				
Load level busy*				U						
Active RUs					R	U	R	W	R	
Load merit					U					
RU select					W					
RU counters							W	R	W	
First time indicator flag							U			
Next RU to run							W	W		
Running RUs									W	U
RU priorities*										R
Task processor capacity										R

*File common to all PCs

R = reads from file
W = writes to file
U = updates file (R & W)

SUPERVISOR PROCESSOR FILE STRUCTURE

MODULE FUNCTIONAL DESIGN SPECIFICATION III

MODULE NAME: Task Processor Executive **Rev No:** 2

AUTHOR: E. Wischmeyer **Date:** 15 July 1980

ONE-LINE DESCRIPTION: This module controls RU execution within the task processor.

DETAILED DESCRIPTION:

Accept messages from supervisor to start RUs or diagnostics or suspend RUs.

"Run" RU or idagnostics.

Send data or results to supervisor.

INPUTS: Two messages from supervisor; data messages for RUs from other RUs; inactive RU's current data

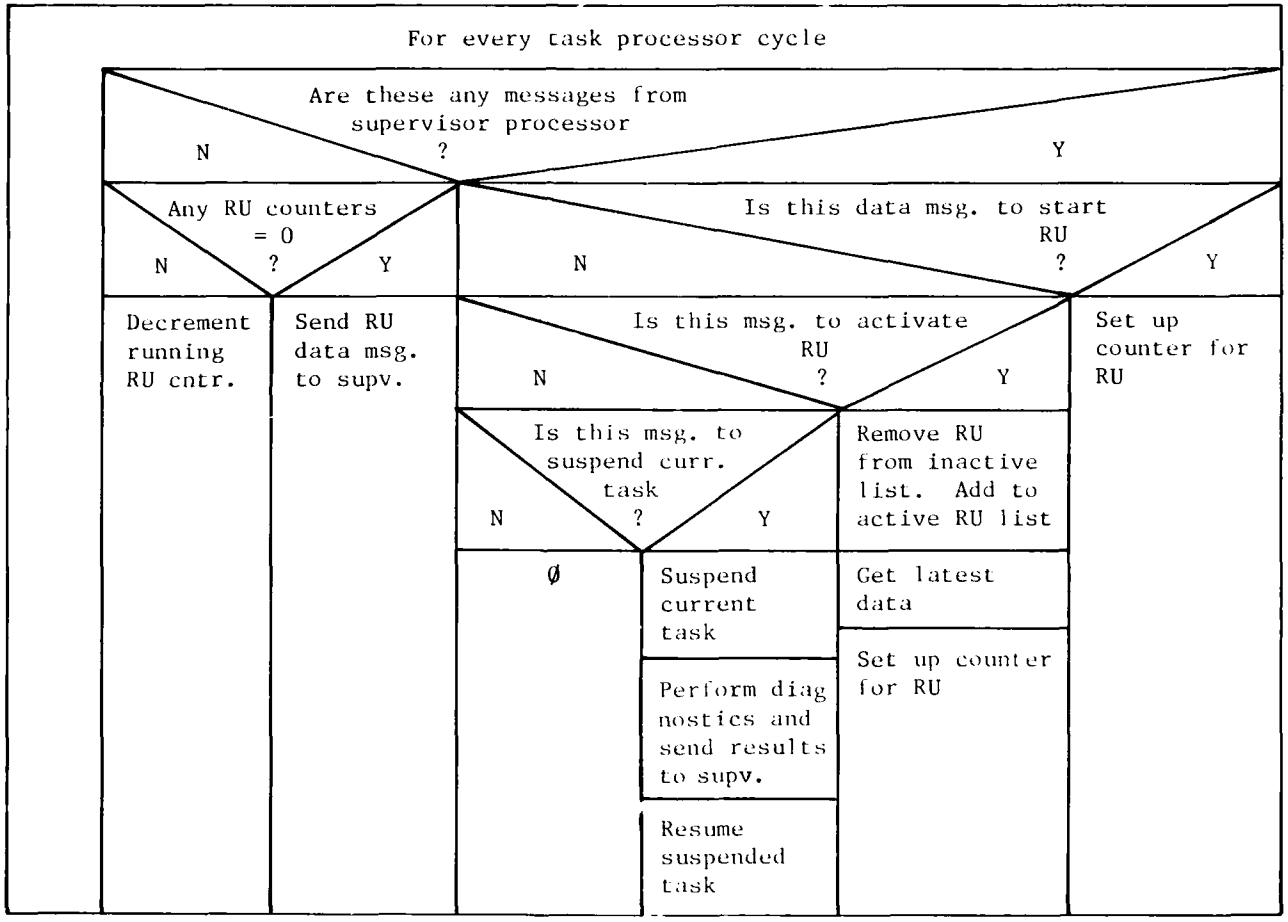
OUTPUTS: Data messages for RUs; results of diagnostics

THIS MODULE CALLED BY: Manage Processor Functions

THIS MODULE CALLS:

UNDEFINED ASPECTS:

TASK PROCESSOR EXECUTIVE



MESSAGES FROM SUPERVISOR	READ
MESSAGES TO SUPERVISOR	WRITE
RU COUNTERS	UPDATE
INACTIVE RU LIST	READ
ACTIVE RU LIST	WRITE

TASK PROCESSOR FILE STRUCTURE

Appendix C

CONTROL OF COMMUNICATION IN A MULTIPROCESSOR NETWORK*

*Previously published as a Technical Report by Marshall C. Pease under Contract no. F33615-80-C-1014.

Table of Contents

1. INTRODUCTION

This report describes a communication algorithm for a switched multiprocessor network. It is also intended to document the simulation program developed to verify procedures that implement this algorithm.

There has been much work done on the control of communication in a network. The early work of Ford and Fulkerson [1] provides a general theory of network operation and includes an algorithm for determining the maximum flow possible between two nodes and the routing that will achieve that maximum. The more recent work of Merlin and Segall is an example of an approach that recognizes the need to account for possible network failure. Much other work could be cited as well. However, none of this work provides a sufficient answer to the problems addressed here. The multiprocessor environment that we consider here imposes its own problems and values on the algorithm to be used to control communication, and so requires its own approach.

In the work described in this report, there are a number of assumptions made about the nature of both the network and the application programs run on it. The assumptions listed below strongly influence the selection of the control algorithm.

- We assume a switched, lattice-type network. That is, each processor communicates to restricted set of neighbors. Each processor has the capability of transmitting messages to any one of its neighbors, with the choice being made by the processor itself and executed by a point-to-point transmission. This contrasts with bus-linked, or multibus, systems.

- The links between a processor and its neighbors are bilateral. A processor, or center, can receive messages without relay only from these neighbors.[1]

- The total number of processing centers is not inordinately large but still considerably exceeds the number of neighbors of any one center. Most messages will need to be relayed. The typical network may have between sixteen and a few hundred processors. A processor may have perhaps four to eight neighbors.

[1]In this report, the terms "processor" and "center" are used interchangeably. In a more general context, we would restrict the former term to units with full processing capabilities. Other units, such as I/O ones with sharply limited capabilities but still able to receive, send, or relay messages, would be counted among the network's centers.

- The connectivity of the network is high, so that there are many potential routes between pairs of centers.

A typical configuration for a network might be in a (4 x 4 x 4) cubic lattice or an (8 x 8) toroidal lattice. If there are centers at all lattice nodes, each center has from three to six neighbors in the former case, four neighbors in the latter. However, it is not necessary that all nodes of a lattice be occupied. The algorithm must also allow for failures of either isolated centers or of large sections of the network. It must be able to maintain communication even after massive damage, provided only that the network remains connected.

The application program is also assumed to have certain properties:

- It is assumed to be composed of a possibly large number of nearly autonomous program modules, each of which is small enough to be handled entirely within a single center of the network. It may be possible for a single processor to handle several modules.

- Each module holds all data and other values required for its own functioning, so that different modules do not access common memories. All coordination among the modules is the result of the transfer of messages between them.

- No data or other values need to be replicated in different modules. If a module must know a value held by another, the inefficiency involved in sending a message asking for the value will not be prohibitive.

- Relatively small amounts of information will be contained in any single message. This is more or less implied by what has been said above, but is sufficiently important to be stated explicitly. It is not intended or expected that any files will be moved. Specific values, or items of information, will be requested and supplied by messages.

- We expect that the program modules will operate asynchronously although this is not strictly essential. We also expect their operations to be nonterminating, although exceptions can exist.

- The volume of communication among the modules required to maintain operations is not large. In general, each module is expected to be computation-limited, so that its speed of operation will not be limited by the communication processes.

- The rate at which data must be inputted to the network, or outtputed from it, is not large and does not represent a severe load on the communication facilities.

These assumptions limit the application environments to which the

architectural approach can be reasonably applied. On the whole, we envisage a system that is programmed with the M-module methodology we have developed and described elsewhere, [3-5], although other methodologies would also be applicable.

We emphasize that the assumptions made here about the application environment and the programming methodology are not universally applicable. The fact is that no single programming methodology can be applied universally. However, there are important applications for which the assumptions are appropriate and the approach applicable.

A major advantage of the approach, when it is suitable, is that it provides for a system that can be highly fault-tolerant, even able to survive massive damage. To achieve this ruggedness, the program modules must be able to move within the network to avoid faults and damage. The algorithms that control such moves, and that maintain the checkpoint data and other information needed to allow restart after a failure, are not our concern here. What does concern us is the need to maintain or quickly restore intermodule communication after unexpected changes in the assignment of modules to processors.

Our general intention is to arrange for the program modules to operate on a virtual machine that is supported by the actual network. That is, the modules should be able to operate as if the virtual machine were the actual one. If the supporting network is damaged or changed for any reason, the support of the virtual machine must change so that a module can continue to send messages to other modules without having to know that a change has occured. The actual transmission of the message in the real network is handled by communication procedures that are resident in the network, or are part of the network's operating system, not of the application program. The details of this transmission process must be invisible to the program modules themselves.

The idea that the communication procedures should be part of the facilities supporting a virtual system is, of course, not new. Even in general-purpose, sequential machines, communications between the CPU and the various memories and I/O devices are controlled by the operating system, not the application program. The application program will usually not even have access to the physical address being used, only to a virtual address that must be interpreted by the operating system. As another example, the choice of routes

in a distributed network such as the ARPANET, or even in the ordinary telephone network, is invisible to the user. There are, however, certain features of the environment envisioned here that impose somewhat unusual conditions. In particular, the following requirements apply or are assumed:

1. Reliability and speed of adaptation are of overriding importance. When a change of configuration occurs, perhaps without warning, the communication facility should adjust to the change with minimal delay.

2. Efficiency of communication is not a prime consideration. While transmission costs cannot be ignored entirely, it it is not a prime requirement that messages be transmitted by the shortest possible paths. It is much more important that every message reach its goal without undue delay, and that, after a fault or damage, the system recover operability as soon as possible.

3. The overhead involved in controlling communications is a potentially serious problem. Most messages will be short but a relatively high density of messages may have to be handled. We must be careful to avoid imposing a heavy computational or memory load with the procedures used for message control.

4. The control procedures should be both locally controlled and locally effective. This means that each center uses only information about its own state and that of its immediate neighbors. Furthermore, it should control only its own actions and, at most, those of its immediate neighbors.

Requirements (1) and (2) are consequences of assuming the set of program modules to be computation-limited. The asynchronous, nearly autonomous nature of the computations, plus the fact that any single message is not likely to contain a large amount of information, makes the efficiency of communication less important than other factors.

Requirement (3) is a consequence of the fact that each message is likely to contain only a small amount of information. We must avoid allowing the control functions to use more network resources than do the messages being controlled.

Requirement (4) is one that we believe to be generally important for any functions used for the control of switched multiprocessor networks. The basic need is to avoid depending on global information or decisions that would have to be disseminated throughout the network. The distribution of global information or commands would be sensitive to faults and other events, and could result in different interpretations by different parts of the network.

If this happened, different parts of the network could take inconsistent actions that could lead to deadlock, to messages circulating indefinitely, or to other disastrous effects. To forestall such consequences, it is very desirable that all control algorithms be locally controlled and locally effective.

The procedures described in this report conform to these assumptions and requirements.

2. PRELIMINARY CONSIDERATIONS

In this section we elaborate on some implications of the assumptions listed in the previous section. We consider first what local information we want to use. We consider how to construct unique identifiers for messages that will make it unnecessary to depend on a global ID bank. We next identify the types of messages that need to be taken into account and the possible modes of their transmission. In the next section we shall examine the operational procedures that can be used.

2.1 Locally Available Information

We first consider what information is available at a given center that can and should be retained there, paying particular attention to the requirement that the algorithm be locally controlled. The available information includes the following:

1. Linkage information. This identifies the nodes in the underlying graph to which the center is connected. In the simulation the list is called the LINKS list. It does not change as faults or other events affecting availability occur. It describes the underlying lattice, not necessarily the network as it actually exists.

2. Neighbors. These are the centers presumed to be nonfaulty to which a given processor is directly connected. In the simulation this list is called the OK.NBRS list. It is a subset of or identical to the LINKS list. While the LINKS list describes the potential of the chosen lattice, the OK.NBRS list describes the actuality of the network as it exists at a given time.

3. Local program modules. These are the program modules currently active in the processor. In the simulation this list is called the LOCAL list. A given module is in the LOCAL list of only one center. These lists for the various centers can be said to provide the boundary condition for the connection between the real and virtual networks.

4. The identities of the messages that have passed through the center. To minimize overhead, each message has a unique ID associated with it. The processor need store only the IDs of the messages, not their contents. This information is necessary to prevent messages from circulating indefinitely in the network.

5. Route information. In the simulation this information is contained in an entry called ROUTE. This information in a given center will specify how a message addressed to a given module should be relayed--i.e., the neighbor to which it should be sent. The route information is the best available information to control the way a requirement in the virtual network should be translated into action in the real one. However, it is quite possible that the current

route information is incomplete or even incorrect. Therefore, the control procedures cannot depend entirely on the existing route information.

The principle of local control would not be violated if we required a center to maintain copies of the various lists (LINKS, OK.NBRS, LOCAL, message IDs, and ROUTE lists) of all its neighbors. However, doing so would substantially increase the overhead load, requiring a great deal of communication between neighbors. We have felt that the benefits would not outweigh the cost. In the simulation, we have not used such information about neighbors or made it available.

2.2 Message IDs

We need a practical way by which each center can generate message IDs that will be globally unique. Each center must be able to determine whether it has already seen a message, and, if so, to refuse to relay it. This is necessary to prevent messages from propagating indefinitely within the network. This could be done by storing copies of the messages a center has seen, but this seems wasteful of storage. Instead, if each message has a unique ID, each center can easily keep track of the IDs of the messages it has seen, refusing to accept duplicates.

To provide a local mechanism that will still lead to globally unique IDs, the simulation uses the following convention:

1. Each processor has a unique number that identifies it and its node in the network's underlying lattice.

2. Each processor stores an integer as the value of the property called NEXT.ID in that center.

3. When a message is being originated by a module local to a center, the message's ID is generated by the center, not the module. (However, a module can include its own identifier as part of the message. If so, this identifier is entirely distinct from the ID considered here.)

4. The ID of a message is generated by concatenating the center's number, a dash, and the value of its NEXT.ID. The center's NEXT.ID is then increased by one. To illustrate, let us suppose a module in processor #4 is generating a message, and the value of #4's NEXT.ID is 7. The ID generated for the message is 4-7 and the NEXT.ID of #4 is set to 8.

5. The ID of a message is preserved during any relay operation.

6. An acknowledgment of a message is not considered a new message, but is given the same ID as the message being acknowledged.

With this convention there can be a problem as messages accumulate, requiring ever higher values of the final component. To avoid this, a maximum value could be established, restarting each NEXT.ID at unity after the maximum value has been reached. The value should be large enough to avoid any possibility of having two messages with the same ID in the system. Such a limit should not be difficult to determine provided the following rule is enforced: before a processor that has been found to be faulty is accepted as recovered, any messages it is holding for transmittal should be erased. A processor found to be temporarily faulty must not be allowed to retain old messages with possibly obsolete IDs.

2.3 Types of Messages

The message-handling procedures must handle a number of types of messages used for different purposes. Those that have been identified as necessary, along with the codes used for them in the simulation program, are as follows:

ORIGINAL — An original message generated by one module for transmittal to another or all other modules.

C-ORIG — An original message generated by a center for transmittal to all centers. We have found no need for center-to-center messages, except the local ones discussed later.

ACK — An acknowledgment by its target module of an ORIGINAL message that is returned to the initiating module. This is not an answer to the original message; it only confirms receipt of the message at its destination.

C-ACK — An acknowledgment of a C-ORIG message. The return will list the centers receiving the message. The expectation is that a C-ORIG message would be used, for example, to assess remaining capabilities after massive damage. The C-ACK acknowledgment is designed to return the information needed in these circumstances.

CTR-STATUS — A broadcast center-to-centers message describing the status of a particular center. It is used in the simulation only to announce that a center has been found to have failed. It is generally available, however, for propagating any information about the status of a single center (not a module) to all centers.

MSG-STATUS The status of an identified message. Its current use is to advise the source of a message that the message may have been prevented from reaching its destination. The originator can then check whether it has received an acknowledgment. If not, the initiator may be able to reinitiate the message in a different mode that will not experience the same trouble.

In addition, there are three types of local messages using what we call the local channel:

L-ACK For local acknowledgments

L-REF For local refusals

C-ACK For responses to C-ORIG messages.

The significance and use of the local channel are discussed later.

2.4 Modes of Message Transmission

We currently distinguish four modes of transmission. Their use depends on the type of the message and the information a processor has available. Those currently recognized as necessary and the codes used to identify them in the simulation program are as follows:

BDCST The message is sent simultaneously to all neighbors of a center except the one from which the message was received.

DIRECTED The message is sent to a particular neighbor either as the destination or for further relay.

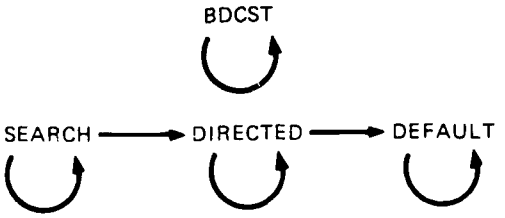
SEARCH This mode is implemented like the BDCST mode, but there are significant differences in when the mode is used, and under what conditions it can be changed. These differences are discussed shortly.

DEFAULT This is implemented like BDCST and SEARCH, but has some further differences in when it is used that are discussed below.

The BDCST mode is used when no destination module is specified. It does not make any use of route information, and does not check the message's destination against a center's LOCAL list.

An ORIGINAL type message that is addressed to a named module will use the SEARCH, DIRECTED, or DEFAULT modes. Both the SEARCH and DEFAULT mode currently use a broadcast transmission method. The differences between them have to do with what a center knows about its route information, and affects how the mode can change. In particular, the SEARCH mode is used when route information is missing, the DEFAULT mode when it may be present but there is evidence that it is faulty.

Module-to-module messages will usually be launched in either the SEARCH or DIRECTED modes. If the center at which it starts has route information, the DIRECTED mode will be used. Otherwise, the SEARCH mode is used. However, a message propagating in the SEARCH mode may reach a center at which route information is available. If so, the mode is changed to DIRECTED for the sake of efficiency. A message in the DIRECTED mode which reaches a center without route information, is relayed in the DEFAULT mode. The SEARCH mode is not used because the inability to continue the DIRECTED mode demonstrates that there must be errors in the network's route information. Once the message starts to use the DEFAULT mode, no further change is permitted, even if the message again reaches a center that has route information, since this information may be erroneous. In summary, the rules governing transitions between modes are as follows:



We also permit a message to be originated in the DEFAULT mode, although this has not been included explicitly in the simulation. That mode would be used if it were necessary to enforce a broadcast kind of transmission while seeking a designated module. This might be done, for example, after receiving a MSG-STATUS report warning of potential trouble, coupled with the absence of

an acknowledgment. The reissue of the message in the DEFAULT mode would ensure that the message will get through if possible--i.e., if the network remains connected.

It is worth noting that there are other ways of implementing the SEARCH and DEFAULT modes. For example, in the DEFAULT mode we might have the center query its neighbors first, before starting a general broadcast. This might have advantages under conditions when a module is most apt to move to a neighbor. Whether these conditions apply depends on the algorithm being used to control the module assignments. Even if the conditions are true, however, the value of the strategy depends on a comparison of the cost of transmitting the message unnecessarily versus the cost of querying the neighbors. We have used the simplest assumption, here, but this does not imply that a more complex definition of the SEARCH and DEFAULT modes might not be useful under appropriate circumstances.

The combinations of TYPE and MODE that have been identified as useful are the following:

#	TYPE	MODE
1	ORIGINAL	BDCST
2		SEARCH
3		DIRECTED
4		DEFAULT
5	C-ORIG	BDCST
6	ACK	DIRECTED
7		DEFAULT
8	C-ACK	DIRECTED (local channel.)
9	CTR-STATUS	BDCST
10	MSG-STATUS	DIRECTED
11		DEFAULT

Others may be added later as additional needs are recognized.

The C-ACK type is included in the table as part of the process initiated by a C-ORIG message. However, C-ACK messages are transmitted on the local channel, discussed later. In fact, all local messages are effectively transmitted in a directed kind of mode, i.e., sent to a particular neighbor.

3. OPERATIONS

In this section, we describe the algorithm in general terms, discussing the reasons for our approach and the particular features that have been emphasized. The actual simulation program is documented in the appendix.

3.1 Basic Operations

The basic operations discussed here are those that transmit module-to-module communications when the network is behaving as expected. Later we shall discuss the variations on this basic set of operations that enable the system to detect and respond to failures.

These operations depend on two buffer systems between a processor and its neighbors. Called the QUEUE and L.QUEUE (local queue) lists in the simulation, these buffers remain available at all times. The QUEUE buffers are used for messages, the L.QUEUE for control signals that are described later. The buffers allow the actual transfer between processors to take place without regard for the state of the receiving processor and without requiring an interrupt or other synchronization device.

Clearly, in a multiprocessor environment the operations of the buffer cannot be totally free. There will usually be several neighbors, all of which can put messages into a processor's buffer. Conflict between neighbors must be avoided, probably giving access to a buffer to only one neighbor at a time. This, however, is simple to arrange; a single bit can be used to lock out all but one neighbor at a time. Because this mechanism does not have to use any of the processor's internal logic, it need not interrupt any other operations.

A similar approach can be used to prevent malfunctioning because of the limited capacity of the buffer. Unless this capacity is excessive, a neighbor can be allowed to make an entry only when the buffer has space available. This can be controlled by a flag bit that indicates available room. The logic to control this bit can be integrated directly into the buffer, capable of acting independently of the rest of the processor. There is no need to interrupt the main operations of the center whose buffer is being controlled.

Let us now suppose that a message has been launched and needs to be relayed from processor #m to #n. Processor #m does so by putting a copy of the message into #n's buffer--in the simulation, by adding the message to the list that is the value of the property QUEUE in #n. Center #n periodically

examines its QUEUE. If there are any messages in it, they are transferred to a list called IN.PROCESS in the simulation program. This transfer is done unconditionally, without any screening of the message for its acceptability. At the time #n makes the transfer, it sends back to center #m a local acknowledgment, a L-ACK type of signal, which is added to #m's L.QUEUE buffer. The purpose is to indicate to #m that #n has noticed the message, thus protecting against the complete failure of #n. Otherwise, the failure of center #n to respond at all might not be noticed, which could cause considerable confusion in the overall process. The detailed use of the local acknowledgment device is discussed later.

In the simulation a message is simply added to the QUEUE or L.QUEUE list of a center, as appropriate. In the design of a multiprocessor system, we would need to consider whether these two operations should use the same interprocessor links. Although this is an implementation aspect that we do not address here, it is worthwhile to visualize these two connections as using different hardware. The two channels are distinct--at least conceptually and functionally--and are used for quite diverse purposes. The local channel is used only for control between centers, the messages transmitted on it being invisible to the program modules. The local channel is used only to pass control information between neighbors, advising a center of events in its neighbors that are significant to it. Furthermore, the control signals can be sufficiently limited in content and form so as to minimize the bandwidth they need. We expect that a single wire would be sufficient in most cases, with local messages being sent in bit-serial form. However, whether they are implemented separately or not, we still speak of the main and local paths as being distinct channels between neighbors.

We shall return later to the use of the control signals on the local channel. They do not have any visible effect in the absence of a problem.

The messages in the IN.PROCESS list of a center are examined periodically. The first action on an IN.PROCESS message is to decide if the message should be processed or not. It will not be processed if the center has already seen a message or acknowledgment with the same ID. If the message is not going to be processed, it is simply abandoned. (There is an exception to this statement discussed later when we consider C-ORIG type messages.) Otherwise its ID is recorded, so that no subsequent copy will be accepted for

processing.

If a message is to be processed, the processor first ascertains whether the destination of the message is a local module. If so, the center accepts the message and initiates an acknowledgment that will be sent back to the originating module. Otherwise the processor prepares to relay the message.

As has been indicated, there are three modes used for module-to-module messages, SEARCH, DIRECTED, and DEFAULT. SEARCH and DEFAULT both use a broadcast procedure, sending copies of the message to all valid neighbors except the one that was the immediate source of the message. The mode of transmission can change during a relay operation. If the mode in which the message was received was SEARCH, the center checks to determine if it has route information to the target. If so, the message is relayed according to that information in the DIRECTED mode. If no route information is available, the SEARCH mode is continued. If the message was received in the DIRECTED mode and route information is available, that mode is continued. Otherwise the mode is made DEFAULT. If the message was received in the DEFAULT mode, it remains in that mode regardless of the availability of route information.

As a message propagates through the network, every processor that sees it uses it to update the center's route information. For example, let us suppose that a message originated by module P has been sent to #n from #m. Processor #n can now record that a valid route to P is through processor #m. Note that there is no guarantee that this route is indeed the best available. Nevertheless, it is probably a reasonably good one. After all, the message did reach processor #n following this route. Other copies that may be around. In any case, the route is a valid one, given the existing state of the network, and so is entered into the center's route information.

When the message finally arrives at the processor that holds its target module, it is accepted and an acknowledgment (type ACK) is initiated with the same ID. This message propagates back is along the route on which the initial message propagated. The processor that initiates the acknowledgment always has up-to-date route information--that derived from the message being acknowledged. Hence the acknowledgment will always be launched in the DIRECTED mode. It can default if a processor along this route should fail, or if the originating module should be moved before the acknowledgment is

received. An acknowledgment, therefore, can drop down to the DEFAULT mode, but it never in the SEARCH mode.

As the acknowledgment propagates back, its ID must be treated slightly differently than was that of the message. The fact that the original message passed through a processor must not prevent that center from relaying the acknowledgment. Therefore, in checking the ID of an acknowledgment, the center determines whether it has seen an acknowledgment with that ID, not merely a message. To permit this control, two lists of IDs are maintained in the simulation. The ID of a message is added to a list called ORIG when an ORIGINAL-type message is seen. When the ID is seen on an ACK-type message, the ID is removed from the ORIG list and added to a list called ACK.

As the acknowledgment is transmitted back to the original source, the relaying centers again pick up route information from it-- information about how best to send messages to the original target module.

Finally, when the acknowledgment is received by the original source of the message, processing is complete. We assume there is no need to acknowledge the acknowledgment.

It may be noted that there is nothing to prevent the original message from propagating further after an acknowledgment has been launched and while the acknowledgment is propagating. If the message is in the SEARCH (or DEFAULT) mode, it is being broadcast. The relay of a broadcast message will be stopped if an acknowledgment of the message has been seen, but only in a single center. If the acknowledgment is transmitted in the DIRECTED mode, it is not being widely distributed. Therefore the message will continue to flood those parts of the network that are not on the path being used by the acknowledgment.

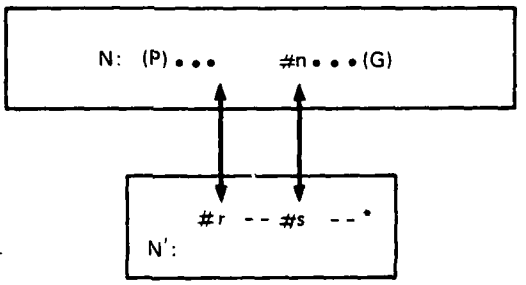
If the messages were much longer than the acknowledgments, it might be desirable to use a different strategy. We could use the broadcast mode for the acknowledgments. This would minimize unnecessary propagation of the message. Since we assume the messages are short, however, we have not used this strategy in the simulation.

3.2 The Failure Problem

Here we discuss the implications of a failure. It is somewhat surprising to find that, even if the system knows of the existence of a failure, there is still potential for trouble. It is important that our procedures take sufficient account of such contingencies.

The main operations will usually not be disrupted by known failures. Suppose center #n is known to have failed. When its failure was detected, a CTR-STATUS message, described later, will have been broadcast. This message will cause all the neighbors of #n to delete #n from their lists of OK.NBRS. If, now, a neighbor of #n, say #m, wishes to send a message in DIRECTED mode through #n, it will recognize that this cannot be done. It will accordingly change the mode of the message to DEFAULT and continue to transmit the message in that mode.

While this procedure will work most of the time, there are some rather special circumstances in which it will fail. As an example, let us consider the following situation:



The box labeled N is the main section of the network. N' is a second section connected to N only through centers #r and #s. Center #s is connected only to center #n in N. There is at least one other center in N' that is connected to #s. Modules P and Q are local to different centers in N. A DIRECTED path from P to Q has been established that goes through #r, #s, and #n, in that order. Suppose, now, that #n fails, and that this is known by #s. As the fact of this failure becomes known in the network, the entry #n is

removed from all of #n's neighbors. However, this does not affect the route information in #r or in earlier centers on the DIRECTED path from P to Q. When a message is sent from P to Q, it will be sent along the established path until it reaches #s. Center #s knows that #n is faulty and so concludes that its route information is in error. It therefore changes the mode to DEFAULT and sends the message on. Since there is at least one other node in N' connected to #s, #s believes the DEFAULT propagation will succeed. However, the DEFAULT version cannot ever reach a center in N because it would have to pass through #r. Center #r, however, has already seen the DIRECTED version, and so will refuse to process the DEFAULT version of the same message. Therefore the DEFAULT version never reaches a point from which it could find a new route to Q. The process fails.

We could introduce other control mechanisms that would allow recovery from this situation--for example, by allowing #r to accept the message in the new mode, or by changing the ID when the mode is changed to DEFAULT. However, the situation arises only under quite special and uncommon failure conditions. Therefore the best approach has been simply to warn the originating module of the possibility of trouble. If the DEFAULT mode version of the message goes through, the originating module will receive an acknowledgment from Q and can thus ignore the warning. Otherwise the originating module can reinitiate the message, but do so in the DEFAULT mode from the start. This reinitiated message will get through if there is any remaining connection between P and Q.

What we need now are the processes that will identify the potentially harmful situations, and those that, if trouble occurs, will cause the system to recover.

3.3 Known Failures: CTR-STATUS-Type Messages

Once a failure is known, whether the knowledge is obtained from the communication process itself or from other tests on the centers, the information should be propagated to all the centers that may need it--i.e., the neighbors of the faulty center. They need to delete the faulty center from their OK.NBRS lists.

As the information to be held in each center has been specified, the only center that knows all the neighbors of a given center is the given center

itself. If this center is faulty, we cannot use its list of neighbors, at least not dependably. Therefore, when a faulty center is detected, the information is broadcast throughout the network. This information will be picked up and used only by those centers that are neighbors of the one found to be faulty.

This procedure causes some unnecessary passing of messages. The alternate, however, would require storing and maintaining much more information about the network in each center. In particular, we would need to store knowledge about the neighbors of each neighbor, plus some information (e.g., route information) about how to reach them. Or else we could refrain from sending the information at all, letting each neighbor of a faulty center find it out for itself. Under the conditions assumed here, neither of these possibilities has seemed cost-effective.

To send information of this kind through the network, we have defined the message type called CTR-STATUS. These messages are originated by centers, not modules, and are broadcast to all centers. No acknowledgment of a CTR-STATUS type message is needed or expected.

A CTR-STATUS-type message could transmit any desired information about the status of a center. It could be used to announce that a previously failed center has become operable or that a new center has been added to the network. It could be used to describe load conditions, or to warn of other possible bottlenecks or difficulties. It is used in the simulation only for failure messages, but the potential significance and application of the type are much broader.

3.4 Detection of Hidden Failures

In the previous section, we discussed means for propagating and using information that identifies a failed center. This information may have been developed by various test procedures executed by the network's centers; such tests will probably be made at suitable intervals. However, since failure can occur without warning, there will be some interval of time during which the failure will not yet have been detected by routine tests.

The failure modes that are most likely to upset the communication control procedures are those in which the message seems to fall into a black hole. If we are not careful, the procedures could be hung up by such an event: the

centers, unaware that a difficulty exists, would wait indefinitely for results that are not going to be provided. Since we allow a neighbor to put a message into a center's QUEUE without interrupt or other simultaneous event, the message could simply sit there indefinitely.

To avoid being hung up waiting vainly for results from a failed center, we provide a test enabling a center to be sure that a neighbor is actually doing something with a message. The L.QUEUE link provides this test. We make a center transmit a local acknowledgment back to the immediate source of the message. This signals that the message has been removed from the QUEUE and transferred to IN.PROCESS status, thereby validating that something is happening.

To make effective use of this check, each processor must keep a record of where it has sent copies of a given message, so that local acknowledgments can be checked off as received. Failure to receive a local acknowledgment from a processor to which a copy has been sent is taken as evidence that that processor has failed.

In the simulation, an action record is kept as a list maintained by a center as the value of the property called SENT. The entries in SENT are organized according to the ID of the message. In particular, SENT is a list, each term of which is also a list whose lead term is the ID. If the message is broadcast, the subsequent terms of a sublist are a list of the neighbors to which the message was sent. If it is sent in DIRECTED mode, the subsequent term is the particular neighbor to which the message was sent.

A processor examines its L.QUEUE list after it has transmitted copies of a message to its neighbors and all local acknowledgments have presumably been received. A local acknowledgment identifies the message by its ID number and the acknowledging processor. When all acknowledgments have been processed, the sublist of SENT under the message's ID is examined. If it is empty, nothing further is needed except to discard information that was being retained against the possibility of a fault. If it is not empty, CTR-STATUS messages are initiated to advise the network of the failed centers. Depending on the type and mode of the message, the center may take additional actions to recover messages blocked by the failures. In particular, if the type is ORIGINAL and the mode is DIRECTED, the message is retransmitted in the DEFAULT

mode. (There is nothing useful that can be done for a message in any other mode.) In addition, if the type was ORIGINAL and the mode was either DIRECTED or DEFAULT, a warning MSG-STATUS-type message is returned to the originator of the message. As has been stated, the message may have been blocked in a way that is not recoverable through any local action. The originating module can is warned so that it can reissue the message in the DEFAULT mode if necessary.

To be able to take the recovery steps, the message is temporarily stored under BACKUP while it is processed. It stays there until the time has come to examine the SENT list for that message. Once the SENT list has been found to be empty, or after the appropriate recovery operations have been performed, the copy is deleted from BACKUP.

3.5 C-ORIG Type, Network Assessment

The last type of message we have defined is the C-ORIG type. It is intended as a way for a center to evaluate the general condition of the network. It is likely to be used after an event that may have caused considerable damage, when it may be necessary to determine which centers are still operative and still connected into a coherent network.

How a center might be triggered into this kind of assessment effort is outside the scope of this report. We can imagine, however, that a center might suspect the need when it sees CTR-STATUS messages reporting some number of simultaneously FAILED centers. When that happens, a center having the capability of inputing from archive might consider whether to reload the application programs, perhaps in a gracefully degraded form. Before doing so, however, it would originate a C-ORIG message to assess the current state of the network, to evaluate whether reload is needed and, if so, to decide how much degradation to enforce.

However triggered and for whatever purpose, a C-ORIG message is regarded as having been originated by a center, not a program module. It is broadcast and requires acknowledgment. (If no acknowledgment is needed, an ORIGINAL type in BDCST mode can be used. The fact that its originator was a center rather than a module is without significance in that case.)

The acknowledgments of a C-ORIG message are returned along the local channel by what we call C-ACK-type messages. They are local messages since each center may have to wait for further responses after receiving a C-ACK

message. In addition, a center must modify the contents of any acknowledgment before relaying it. At a minimum, it will add itself to the list of operative centers being returned.

As with any other message, a center will refuse a C-ORIG message if it has already seen a copy that was relayed to it from a different center. As before, this is done by checking the unique ID of the C-ORIG message. However, the rejection of an ordinary message (ORIGINAL or ACK type) simply causes it to be abandoned. The refusal to accept a C-ORIG message must be returned so that the source will know that it should not wait for an acknowledgment from the refusing center. Consequently, the center refusing the message returns a L-REF type message to the center from which the C-ORIG message came.

The refusal is sent through the local channel. This happens, if at all, at the start of processing. In contrast, the L-ACK acknowledgment of a message by the same center is sent at the time the message is dequeued, before its ID is even examined. Therefore, the L-ACK message is received first, any L-REF or C-ACK message from the same center later. The check determining that the center's neighbors are at least doing something about the message is completed before any L-REF or C-ACK messages are received.

An operative center will return a C-ACK message under three conditions:

1. None of its neighbors have returned a L-ACK local acknowledgment. The center is at the end of a cul-de-sac in the remaining network. It returns a C-ACK message listing only itself as operative. It cannot report any other center as operative.

2. All neighbors that have sent local acknowledgments have refused to accept the C-ORIG message, returning L-REF messages. The center is a leaf at the end of a branch of the tree along which the C-ORIG message is passing. The center returns a C-ACK message listing itself. Any of its neighbors that are still operative are being reported along other paths. Duplications are to be avoided.

3. All neighbors that have returned a local acknowledgment and that have not sent a local refusal have returned C-ACK messages. The center returns a C-ACK message that lists itself plus the concatenated lists returned by all the C-ACK messages it has received.

While a C-ACK is transmitted along the local channel, it is processed in ways that are somewhat more complex than the other local messages. Therefore,

when a C-ACK message is recognized in the L.QUEUE, it is transferred to the IN.PROCESS list for processing.

The C-ORIG message requires the construction of a SENT list, as used by ORIGINAL-type messages, to detect inoperative centers. The processes that construct this list and use the L-ACK messages to reduce it are the same as before.

A center will also need to check off the L-REF or C-ACK messages as they are received. This will need to be done against a list similar to that under SENT, but it must be a different copy. Besides, the center cannot know how long it must wait for L-REF or C-ACK messages. The C-ACK message from a neighbor will not be returned until that neighbor has accounted for all of its neighbors. Hence the checkoff list may need to be retained for some time. In the simulation this list is constructed similarly to the SENT list, but under the property name WAIT. Its entries are deleted whenever an L-REF or C-ACK message is received. A center generates a C-ACK message when its WAIT list has been reduced to whatever remains of the SENT list. This indicates that one of the three conditions cited has been fulfilled.

It is also necessary to store the lists that have been returned by C-ACK messages to a center until that center is itself ready to generate a C-ACK message. The information accumulated as a simple list by appending the lists in the C-ACK messages that have been received. Conceivably, however, there could be several C-ORIG messages propagating at one time, perhaps originated by different centers. Consequently, the accumulating list of acknowledging centers is stored under the ID of the C-ORIG message, and these lists stored as values of the property named LIST in the center.

The immediate source of the C-ORIG message also needs to be retained, so as to know where to send the C-ACK message when the time comes. We could store copies of the message itself under BACKUP as before, but that seems undesirable in this case. (In the previous situation there are conditions under which the message must be locally reissued in a different mode. Here we never need more than the immediate source.) The storage is done under the property name SOURCE, here too by ID.

Finally, the WAIT, LIST, and SOURCE entries can be deleted at the time the center returns a C-ACK message.

When the originating center has received C-ACK or L-REF messages from all of its neighbors that are sufficiently active to return L-ACK messages, the center knows all the centers that are still operative and to which it is connected by paths through operable centers. It has the information needed to assess the condition of the network.

3.6 CONCLUSIONS

The procedures described here enable a network to maintain intermodule communication in the presence of unexpected faults or other events, even massive damage. They allow communication to proceed with reasonable efficiency, as long as suitable and correct route information is available. Messages are not lost, although they may be slowed down, if route information is unavailable or in error because program modules have been moved within the network. If either of these conditions occurs, continued use of these procedures will fill in the missing route information or will correct its errors.

A simulation system has been coded for operation on a computer system and functions correctly under the failure scenarios that have been examined. The variety of circumstances that can arise, however, is huge. We cannot be absolutely certain that all contingencies have been accounted for. It would be desirable to apply proof techniques to the control algorithm to establish its validity under all conditions.

A number of variations on the particular algorithm developed here are possible and might be advantageous under slightly different conditions. For example, the route information that is developed does not necessarily lead to the shortest possible route from one module to another. As the target module is moved around the network, the modified route can become suboptimal. This could be significant if the network were large and contained highly redundant communication resources. Under these conditions, it might be worthwhile to introduce additional modes that could force the system to seek an improved route.

Other variations could also be considered, and perhaps should be if the assumptions that have been made are altered. For example, we have assumed that the messages are all short. If they can be long, there may be significant advantage in using a broadcast mode for the acknowledgment, so as

to cut off extraneous copies of the message as soon as possible. Each of the stated assumptions implies certain characteristics of a desirable control algorithm. When the assumptions are changed, the implications should at least be reconsidered.

Given these caveats, however, the algorithm developed here does demonstrate a way to control communication in a multiprocessor network of the type considered. Since this environment is quite different from the more typical communication networks, it has been necessary to develop a control algorithm that fits the particular requirements of the application. The work described shows that it is possible to meet the conditions that apply in such an environment.

REFERENCES

[1] L. R. Ford, Jr. and D. R. Fulkerson, "Flows in Networks" (Princeton University Press, Princeton, New Jersey, 1962).

[2] P. M. Merlin and A. Segall, "A Failsafe Distributed Route Protocol," EE Pub. No. 313, Laboratory for Information and Decision Processing, Massachusetts Institute for Technology, Cambridge, Massachussetts (September 1978).

[3] M. C. Pease III, "ACS.1: An Experimental Automated Command Support System," IEEE Trans. on Systems, Man, and Cybernetics, Vol. SMC-8, No. 10, pp 725-735 (October 1978).

[4] C. A. Monson, P. R. Monson, and M. C. Pease "A Cooperative Highly-Available Multi-Processor Architecture: CHAMP," Proceedings of Compcon 1979, Washington, D. C., pp 349-356 (September 1979).

[5] M. C. Pease "M-Modules: A Design Methodology," Technical Report 17 (Contract No. N00014-77-C-0308 with the Office of Naval Research, Department of the Navy, Arlington, Virginia) SRI International, Menlo Park, California (March 1979).

Appendix

SIMULATION SYSTEM

In this appendix, we describe and document the simulation system. We list the functions that are used in the simulation, and discuss their purposes and interactions. The organization of the appendix is as follows:

1 Network Representation

2 Formats

3 Functions

- a. Utility functions
  - a1. Functions for the management of a.lists
  - a2. Output and trace functions
  - a3. Network management functions
- b. High level functions
- c. Broadcast messages
- d. Module-to-module messages
- e. Message acceptance
- f. Propagation of acknowledgments
- g. Detection of failed centers
- h. CTR-STATUS type messages
- i. Unblocking messages
- j. MSG-STATUS type messages

k. C-ORIG type messages
   k1. Refused messages
   k2. C-ACK responses
   k3. Restart

In each of the various subsections of 3, the functions developed for the required operations are given. The system as a whole runs in INTERLISP under TOPS-20 on a DEC 1090T computer facility.

1 NETWORK REPRESENTATION

The network is represented by a list called C.CENTERS. An example is given below. This shows the C.CENTERS list for a (2 x 2 x 2) network loaded with modules labeled from A through R. The omitted parts for centers #2 through #8 are the same as for center #1.

```
C.CENTERS =
    (DEFAULT (1 (LINKS 2 3 5)
                (LOCAL R M P)
                (NEXT.ID . 1)
                (ROUTE (R . 1)
                       (M . 1)
                       (P . 1))
                (OK.NBRS 2 3 5)
                (ORIG)
                (ACK)
                (IN.PROCESS)
                (QUEUE)
                (L.QUEUE))
             (2 (LINKS 1 4 6)
                (LOCAL J K)
                  . . .
             (3 (LINKS 1 4 7)
                (LOCAL Q H)
                  . . .
             (4 (LINKS 2 3 8)
                (LOCAL F)
                  . . .
             (5 (LINKS 1 6 7)
                (LOCAL A I O)
                  . . .
             (6 (LINKS 2 5 8)
                (OK.NBRS 2 5 8)
                (LOCAL B G L)
                  . . .
             (7 (LINKS 3 5 8)
                (LOCAL C)
                  . . .
```

```
(8 (LINKS 4 6 7)
   (LOCAL D E N)
      . . .
(SIZE . 8))    .
```

The lead term, DEFAULT, is there to prevent the list from being annihilated when its contents are eliminated. In INTERLISP, a list cannot be completely removed by the same functions that can delete terms. The simplest way to avoid difficulties is to include a term that need not ever be deleted.

The list as a whole is a list of lists. Except for DEFAULT and the final term that records the number of processors in the network, each sublist is headed by a number that identifies a processor. The list that is the value of property #1 in the list, for example, provides the information used by processor #1.

Within a sublist the value of LINKS is the list of processors, or locations in the lattice, to which the processor is connected. The value of OK.NBRS, later in the list, is originally the same as the LINKS list. It lists the neighbors that are good processors. Should a processor be declared faulty, a property-value pair (FAILED . T) is added to the center's list. This flag causes all actions to be aborted by the processor. When a center discovers that one of its neighbors has failed, the number of the failed processor is deleted from the center's list of OK.NBRS.

The property called NEXT.ID, which always has an integer value, is initially given the value 1. It is used to generate an identifier for a message as described later.

The property called ROUTE contains the route information available to the processor. It has the format of a list of lists. Each of its components is a list of two elements, the first the name of a module, the second the number of a neighbor. The only route information initially available is for the local modules for which the processor itself is identified as the route. If ROUTE for center #1 contained the term (G . 2), it would mean that a message for module G should be sent to center #2 for relay. Since G is actually in #6, a neighbor of #2, this route information would be valid in the simulated network.

The value of LOCAL is a list of the modules that are present in the processor. If module X is moved from processor #n to #m, X will be deleted from #n's LOCAL list and added to #m's list. In addition, the route entries of both #n and #m are modified, but not the route information in any other center. Hence, the movement of a module from one center to another will make some of the route

information in the network incorrect. This incorrect route information will be corrected as messages continue to be passed through it.

No values are given initially to the other properties named in the list. Their individual significance will be described as their use is considered.

2 MESSAGE FORMATS

The various types of messages and their possible modes have been identified in the body of the report. For the simulation, however, it is necessary to specify formats for all such types. The following formats are used:

a. Type ORIGINAL:

```
(ID (FROM . —) (AT . —) <(TO . —)> (TYPE . ORIGINAL)
    (MODE . —) <(LAST . —)> <(ACK . T)> <(MSG —)>)
```

The general structure is a list of lists that is unordered, except that the list must be headed by the message's ID. The sublists (except for that headed by MSG) are dotted pairs, a convenient INTERLISP structure. They can be read simply as property-value pairs. For example, the value of the property named TYPE is ORIGINAL. The value of FROM will be the name of some program module. Terms that are surrounded by < ... > are optional; they may or may not be present.

In the particular list shown here, the value of FROM is the originating module; the value of AT is the center from which the message originated. The value of TO, if present, is the module to which the message is being sent. If TO does not have a value, the message is being broadcast to all modules. The value of MODE will be BDCST, DIRECTED, SEARCH, or DEFAULT. The value of LAST is the center that was the immediate source of the message, i.e., the center that last relayed it. This pair is absent when the message is first created. MSG is the message itself, generally absent in the simulation. The dotted pair (ACK . T) may or may not be present. If present, an acknowledgment (type ACK) is required.

b. Type C-ORIG:

```
(ID (AT . —) (TYPE . C-ORIG) (MODE . BDCST) <(LAST . —)>
    <(MSG . —)>)
```

where the value of AT is the originating center. The property name AT is used here for uniformity with ORIGINAL type messages. This type is originated by a center, not a program module, and so does not have a FROM property.

c. Type ACK:

```
(ID (FROM . --) (AT . --) (TO . --) (TYPE . ACK)
    (MODE . --) <(LAST . --)> <(MSG . --)>)    .
```

The ID of an acknowledgment is the same as that of the message being acknowledged.

The value of FROM is the module acknowledging receipt, while the value of AT is its center. The value of TO is the module to which the acknowledgment is being sent. The values of FROM and TO in an acknowledgment will be interchanged from what they were in the message being acknowledged. The MODE will be DIRECTED initially, but can change to DEFAULT. (Since the message being acknowledged has just been received, there is always route information available in the center originating the acknowledgment. Therefore, the acknowledgment always starts in the DIRECTED mode.) LAST is the immediate source of the acknowledgment, the last center that relayed it.

The possibility of a message is included, since the responding module might need to qualify its acknowledgment--although this possibility is not used in the simulation. However, in an actual system we might use a qualified acknowledgment if, for example, a parity check fails. This would indicate that the available route information is valid, but that the message should be repeated.

d. Type CTR-STATUS:

```
(ID (AT . --)(TYPE . CTR-STATUS)(MODE . BDCST)
    (CENTER . --)(STATUS . --))    .
```

This type is used to broadcast the information that the condition of a center has changed. The message's ID is generated by the center reporting the condition and identified in the message by the value of AT. The center whose changed condition is being reported is identified by the value of CENTER. In the simulation, the value of STATUS is always FAILED, although other values might be used in the system. For example, we might allow STATUS to have the value ACTIVE to indicate a new center or one that has recovered from a fault.

e. Type MSG-STATUS:

```
(ID (AT . --) (TO . --) (TYPE . MSG-STATUS) (MSG-ID . --)
    (MODE . --) (STATUS . BLOCKED)    .
```

This is used to advise a module that a message it originated has been blocked at the reporting center, and so may have failed to reach its destination.

The ID is a new one generated by the reporting center. The value of AT is the center initiating the report. The value of TO is the module that originated the message whose status is being reported. The value of MSG-ID is the ID of the message whose status is being reported. The mode will either be DIRECTED or DEFAULT.

In the simulation the value of STATUS will be BLOCKED, indicating that the transmittal of the message being reported on was blocked at the reporting center. This does not mean that the message did not reach its destination. It could have got there by some other route elsewhere in the network. The MSG-STATUS report simply alerts the original source of the message to the possibility that its message might have been totally blocked. It is up to the originating source to determine if the transmission has actually failed and, if so, what to do about it. The failure to get the message through will be indicated by the absence of an acknowledgment after an appropriate wait. The initiating center can then, if necessary, reissue the message in the DEFAULT mode. It will go through in that mode if the target module is at all reachable.

f. Local messages:

There are also three types of local messages:

f1. Local Acknowledgment:

(ID (CENTER . --)(TYPE . L-ACK)) .

The value of CENTER is the center acknowledging receipt of the message with a given ID. As was indicated in the main text, the local acknowledgment provision is used to give assurance that the acknowledging center is still operational--at least to the extent that it has noticed the message and is about to do something with it.

f2. Refusal of a message:

(ID (CENTER . --)(TYPE . L-REF)) .

Here, too, the value of CENTER is the center at which the message with the given ID has been refused. This type is currently used only in response to a C-ORIG type of message. It could be employed in other circumstances, however, should a relaying center need to know if a neighbor has already seen the message but from a different source.

f3. C-ACK message:

(ID (CENTER . —) (TYPE . C-ACK) (LIST —)) .

As has been indicated, this is the acknowledgment form for responding to a C-ORIG type of message. The value of CENTER is the reporting center. The value of LIST is a list of centers that are known by the reporting center to be operative. It includes the reporting center itself, plus any lists that have been sent to it in C-ACK messages from other centers.

3 FUNCTIONS

a. Utility Functions

The first set of functions that we describe are those that provide general capabilities that are useful in the simulation. There are three broad categories we list: (1) functions that manage what are called a.lists; (2) functions that provide output and tracing capabilities; (3) functions that manage the simulated network, specifying faults and other events.

a1. A.list Management Functions

What are called a.lists, or association lists, are used for representing the network and for other information. An a.list is an unordered list of property-value pairs expressed as lists, often dotted pairs. The value of a named property is retrieved from the a.list by searching for the sublist whose leading term is the property name. This is the reason for the name: the value is retrieved associatively.

To make a.lists useful, we need various functions for manipulating them. The first of these, A.GETP, retrieves the value of a named property from an a.list:

```
(A.GETP
  [LAMBDA (A.LIST PROP)
    (CDR (ASSOC PROP A.LIST])
```

There is also a function, A.GETP#, that is elaborated from A.GETP. To understand its purpose, it must be recognized that the value of a property in an a.list can itself be an a.list. This nested construction can be extended to any depth. A.GETP# retrieves values that may be deep inside a nested a.list. Whereas A.GETP takes the name of a property, PROP, A.GETP# takes a list of property names, PROP.LIST. It works its way down this list until it reaches the end. That is, it takes the first property name on the list and retrieves the list that is the value of that property in the original list. In

the latter, it then retrieves the value of the second property in PROP.LIST. It continues to do this until it reaches the end of the list of property names. If the process fails at any step, then the function fails generally, returning NIL as its value.

```
(A.GETP#
  [LAMBDA (A.LIST PROP.LIST)
    (COND
      ((CDR PROP.LIST) (A.GETP# (A.GETP A.LIST (CAR PROP.LIST))
                                (CDR PROP.LIST)))
      (T (A.GETP A.LIST (CAR PROP.LIST])
```

A.GETP and A.GETP# illustrate two naming conventions. First, the names of all a.list management functions start with "A.—." Second, all functions that use a list of property names to manage nested a.lists are terminated by #.

The functions that set or change the value of a named property in an a.list are

```
(A.PUT
  [LAMBDA (A.LIST PROP VAL)
    (PROG (XX)
          [COND
            ((SETQ XX (ASSOC PROP A.LIST)) (RPLACD XX VAL))
            ((NCONC1 A.LIST (CONS PROP VAL]
          (RETURN VAL])    ,
```

and

```
(A.PUT#
  [LAMBDA (A.LIST PROP.LIST VAL)
    (COND
      ((CDR PROP.LIST) (A.PUT# (A.GETP A.LIST (CAR PROP.LIST))
                               (CDR PROP.LIST)
                               VAL))
      (T (A.PUT A.LIST (CAR PROP.LIST) VAL])    .
```

If the property is already present, A.PUT changes its value to VAL, or to NIL if no VAL is given. If it is not already present, the name is added to the a.list as a property with the indicated value.

If the intermediate properties in PROP.LIST are not already presented in a nested a.list, A.PUT# does not add them. In principle it should, but we have not needed to that behavior, and have not developed the additional processes that would be needed.

The functions that add a property-value pair to an a.list are

```
(A.ADDPROP
  [LAMBDA (A.LIST PROP VAL)
    (PROG ((XX (ASSOC PROP A.LIST)) YY)
          (SETQ YY (CDR XX))
          [COND
            ((NULL YY) (A.PUT A.LIST PROP (LIST VAL))
                       (SETQ XX (LIST PROP VAL)))
            ((LISTP YY) (RPLACD XX (CONS VAL YY)))
            (T (RPLACD XX (CONS VAL (LIST YY]
          (RETURN (CDR XX])    ,
```

and

```
(A.ADDPROP#
  [LAMBDA (A.LIST PROP.LIST VAL)
    (PROG (SUBLST)
          (COND
            [(CDR PROP.LIST)
              (COND
                ((SETQ SUBLST (A.GETP A.LIST (CAR PROP.LIST)))
                  (A.ADDPROP# SUBLST (CDR PROP.LIST) VAL))
                (T (A.PUT A.LIST (CAR PROP.LIST)
                          (LIST (FORM.PUT.CHAIN (CDR PROP.LIST) VAL]
            (T (A.ADDPROP A.LIST (CAR PROP.LIST) VAL])    .
```

A.ADDPROP# will add a nested sequence of property names to an a.list if they are not already present. It uses the function FORM.PUT.CHAIN for this purpose. FORM.PUT.CHAIN is called if the effort to retrieve a sublist as the value of a property in the PROP.LIST fails, indicating that that property was not originally present.

```
(FORM.PUT.CHAIN
  [LAMBDA (PROP.LIST VAL)
    (COND
      ((CDR PROP.LIST) (CONS (CAR PROP.LIST)
                             (FORM.PUT.CHAIN (CDR PROP.LIST) VAL)))
      (T (CONS (CAR PROP.LIST) VAL])
```

A.REMPROP and A.REMPROP# remove a named property, along with any value it may have, from an a.list:

```
(A.REMPROP
  [LAMBDA (A.LIST PROP)
    (PROG (X)
          [COND
            ((SETQ X (ASSOC PROP A.LIST))
              (COND ((EQUAL A.LIST (LIST X)) (RPLACD X NIL))
                    (T (DREMOVE X A.LIST]
          (RETURN PROP])    ,
```

and

```
(A.REMPROP#
  [LAMBDA (A.LIST PROP.LIST)
    (PROG (XX)
          (COND [(CDR PROP.LIST)
                 (SETQ XX (A.GETP A.LIST (CAR PROP.LIST)))
                 (COND ((OR (ATOM XX)
                            (AND (LISTP XX)
                                 (EQUAL (LENGTH XX) 1)))
                        (A.PUT A.LIST (CAR PROP.LIST)))
                       (T (A.REMPROP# XX (CDR PROP.LIST]
                (T (A.REMPROP A.LIST (CAR PROP.LIST])    .
```

If the value of a property in an a.list is a list, a new term can be added to this list by the following:

```
(A.ADDVAL
  [LAMBDA (A.LIST PROP VAL)
    (PROG ((XX (ASSOC PROP A.LIST)) YY)
          (SETQ YY (CDR XX))
          [COND
            ((NULL YY) (A.PUT A.LIST PROP (LIST VAL))
                       (SETQ XX (LIST PROP VAL)))
            ((LISTP YY) (RPLACD XX (NCONC1 YY VAL)))
            (T (RPLACD XX (CONS VAL (LIST YY]
          (RETURN (CDR XX])    ,
```

and

```
(A.ADDVAL#
  [LAMBDA (A.LIST PROP.LIST VAL)
    (PROG (SUBLST)
          (COND [(CDR PROP.LIST)
                 (COND
                   ((SETQ SUBLST (A.GETP A.LIST (CAR PROP.LIST)))
                        ADDVAL# SUBLST (CDR PROP.LIST) VAL))
                   (T   .PUT A.LIST (CAR PROP.LIST)
                          IST (FORM.A.CHAIN (CDR PROP.LIST) VAL]
                (T (A.ADDVAL A.LIST (CAR PROP.LIST) VAL])    ,
```

where FORM.A.CHAIN is invoked if the a.list does not already contain the properties named in the property list:

```
(FORM.A.CHAIN
  [LAMBDA (PROP.LIST VAL)
    (COND
      ((CDR PROP.LIST) (LIST (CAR PROP.LIST)
                             (FORM.A.CHAIN (CDR PROP.LIST) VAL)))
      (T (LIST (CAR PROP.LIST) VAL])
```

If a property is not already present in an a.list, A.ADDVAL adds it. It should be noted, however, that it does so differently from A.PUT. A.PUT adds the property-value pair as a dotted pair, while A.ADDVAL adds it as a list. A.GETP, then, is the reverse of A.PUT, not A.ADDVAL. If A.ADDVAL has been used to add VAL under the name PROP, A.GETP, if called with with PROP, will return (VAL) instead of VAL—i.e., the list whose only term is VAL. This can be confusing if the distinction is not recognized.

If we must remove a single term from a list that is the value of a property, the following functions are used:

```
(A.REMVAL
  [LAMBDA (A.LIST PROP VAL)
    (PROG ((XX (ASSOC PROP A.LIST)))
          (COND [(EQUAL XX (LIST PROP VAL))
                 (COND ((CDR A.LIST) (A.REMPROP A.LIST PROP))
                       (T (RPLACA A.LIST (LIST PROP]
                (T (DREMOVE VAL XX)))
          (RETURN VAL])    ,
```

and

```
(A.REMVAL#
  [LAMBDA (A.LIST PROP.LIST VAL)
    (COND
      ((CDR PROP.LIST)
        (A.REMVAL# (A.GETP A.LIST (CAR PROP.LIST)) (CDR PROP.LIST)
                   VAL))
      (T (A.REMVAL A.LIST (CAR PROP.LIST) VAL])    .
```

These functions provide a basic capability for manipulating a.lists.

a2. Output and Trace Functions

Certain outputting and tracing functions have been defined for convenience. MAPPRINQ is a print function that can execute a possibly extensive list of print operations:

```
(MAPPRINQ
  [NLAMBDA (LST)
    (MAPC LST (FUNCTION (LAMBDA (#X)
              (COND
                ((EQUAL #X (QUOTE TERPRI)) (TERPRI))
                ((STRINGP #X) (PRIN1 #X))
                ((ATOM #X) (PRIN1 (EVAL #X)))
                ((EQUAL (CAR #X) (QUOTE TAB)) (TAB (CADR #X)))
                [(EQUAL (CAR #X) (QUOTE RPTQ))
                   (RPTQ (CADR #X) (EVAL (CADDR #X]
                (T (PRIN1 (EVAL #X])    .
```

Note that this is an NLAMBDA function. This means that it does not evaluate its argument, LST. LST, therefore, is not quoted, but is given directly as a list of terms.

MAPSTRING is a function that converts a list of atoms into a conveniently printable string. For example, it converts the list (A B C) into a string that will be printed as "A, B, C." The string is terminated with a period unless TERMINAL.FLG is given a non-NIL value.

```
(MAPSTRING
  [LAMBDA (LST TERMINAL.FLG)
    (PROG ((LST1 (REVERSE LST)))
          [COND [TERMINAL.FLG (SETQ STR (MKSTRING (CAR LST1]
                (T (SETQ STR (CONCAT (MKSTRING (CAR LST1)) "."]
          [COND ((CDR LST1) (MAPC (CDR LST1)
                                  (FUNCTION (LAMBDA (XX)
                                    (SETQ STR (CONCAT (MKSTRING XX)
                                                      ", " STR]
          (RETURN STR])    .
```

The function, PRINT.EVENT, is used to trace what is happening in the network. If the global variable called TRACE is given a non-NIL value, PRINT.EVENT is called any time a message is dequeued from the QUEUE of any center. It prints the center's identity and the message being dequeued. If the global variable called L.TRACE is set, the same thing happens every time a message in L.QUEUE in any center is dequeued.

```
(PRINT.EVENT
  [LAMBDA (CENTER MSG)
    (MAPPRINQ ("Center = " CENTER TERPRI (TAB 5) "Msg = " MSG TERPRI])
```

PRINT.FAILURES is called whenever a center is discovered to have failed. It is also essentially a trace function, but it is not switchable, although it could easily be made so.

```
(PRINT.FAILURES
  [LAMBDA (CENTER LST)
    (COND ((CDAR LST)
           (MAPC (CDAR LST) (FUNCTION (LAMBDA (QQ)
             (MAPPRINQ ("Center #" QQ
                        " presumed faulty by center #" CENTER
                        "." TERPRI])
```

Finally, there is a list that stores a record of the main events if the global variable H.FLG (for history flag) is given a non-NIL value. The list is called C.HISTORY (for communication history). Storage on this list is done either by ADD.C.HISTORY or ADD.H.STRING.

```
(ADD.C.HISTORY
  [LAMBDA (CENTER MSG DESC LST)
    (RPAQ C.HISTORY (CONS (LIST (CAR MSG) CENTER DESC LST) C.HISTORY])
```

ADD.H.STRING allows one to place an arbitrary string on the history list. This function is used whenever the parametric format used by ADD.C.HISTORY is inappropriate.

```
(ADD.H.STRING
  [LAMBDA (STR)
    (RPAQ C.HISTORY (CONS STR C.HISTORY])
```

The history list can be read directly, but PRINT.C.HISTORY gives a more convenient format.

```
(PRINT.C.HISTORY
  [LAMBDA (M)
    (PROG [(N 0) (LST (CDR (REVERSE C.HISTORY]
          [COND
            (M (SETQ LST (NTH# LST M)) (SETQ N (SUB1 M]
          (RPTQ 2 (TERPRI))
          [MAPPRINQ
            ("Event  Msg.  Acting      Type        Mode           To"
                     TERPRI
          "  #     ID   Center     Action                    Centers"
                     (RPTQ 2 (TERPRI]
```

```
[MAPC LST (FUNCTION (LAMBDA (ZZ)
          (PROG ((DESC (3RD ZZ)) TYPE MODE STR)
                (SETQ N (ADD1 N))
                (COND ((STRINGP ZZ)
                       (MAPPRINQ (TERPRI (TAB 2) N ZZ))
                       (RETURN NIL)))
                (COND
                  ((EQUAL DESC (QUOTE ACK-DIR))
                    (SETQ TYPE (QUOTE ACK))
                    (SETQ MODE (QUOTE DIRECTED)))
                  ((EQUAL DESC (QUOTE ACK-DEFAULT))
                    (SETQ TYPE (QUOTE ACK))
                    (SETQ MODE (QUOTE DEFAULT)))
                  (T (SETQ TYPE (QUOTE ORIGINAL))
                     (SETQ MODE DESC)))
                [COND
                  ((NULL (4TH ZZ)) (SETQ STR "n/a."))
                  [(LISTP (4TH ZZ))
                    (SETQ STR (MAPSTRING (4TH ZZ]
                  (T (SETQ STR (MAPSTRING (LIST (4TH ZZ]
                (MAPPRINQ ((TAB 2) N (TAB 8) (CAR ZZ)
                           (TAB 15) (2ND ZZ) (TAB 24) TYPE
                           (TAB 38) MODE (TAB 50) STR))
                (TERPRI]
(TERPRI)
(RETURN (CHARACTER 127])    .
```

Table 1 illustrates the form in which the history list is printed by PRINT.HISTORY. The particular network used had the connectivity of the binary cube, 2 x 2 x 2. The first command was to send a message from module G in center #2 to module E, which turned out to be in center #7. No route information was available at that time. The message propagated in the search mode until it reached its target module. An acknowledgment was returned in the DIRECTED mode. The G and E modules were then moved to make the existing route information wrong. Another message was sent from module G to module E. It shifted between the search and directed modes as appropriate.

Table 1

A HISTORY OF MESSAGE PASSING
IN A BINARY CUBIC NETWORK

Event #	Msg. ID	Acting Center	Type Action	Mode	To Centers
1	2-1	2	ORIGINAL	INITIAL	n/a.
2	2-1	2	ORIGINAL	SEARCH	1, 4, 6.
3	2-1	1	ORIGINAL	SEARCH	3, 5.
4	2-1	4	ORIGINAL	SEARCH	3, 8.
5	2-1	6	ORIGINAL	SEARCH	5, 8.
6	2-1	3	ORIGINAL	SEARCH	4, 7.
7	2-1	5	ORIGINAL	SEARCH	6, 7.
8	2-1	8	ORIGINAL	SEARCH	7.
9	2-1	7	ACK	INITIAL	n/a.
10	2-1	7	ACK	DIRECTED	3.
11	2-1	3	ACK	DIRECTED	1.
12	2-1	1	ACK	DIRECTED	2.
13	<< Module Shift >> : Module #E shifted to center #8 from #7.				
14	<< Module Shift >> : Module #G shifted to center #6 from #2.				
15	6-1	6	ORIGINAL	INITIAL	n/a.
16	6-1	6	ORIGINAL	SEARCH	5, 8.
17	6-1	2	ORIGINAL	DIRECTED	1.
18	6-1	5	ORIGINAL	SEARCH	7.
19	6-1	8	ACK	INITIAL	n/a.
20	6-1	1	ORIGINAL	DIRECTED	3.
21	6-1	7	ORIGINAL	SEARCH	8.
22	6-1	8	ACK	DIRECTED	6.
23	6-1	3	ORIGINAL	DIRECTED	7.

PRINT.C.HISTORY also allows one to specify that the printout should start at a specified event.

a3. Network Management Functions

Certain other functions have been defined for control of the network during simulation. For example, we may wish to assert that a given center has failed. This is done by means of the following very simple function:

```
(FAILED.CENTER
  [LAMBDA (CENTER)
    (A.PUT# C.CENTERS (LIST CENTER (QUOTE FAILED)) T])
```

All this does is to add the property-value pair (FAILED . T) to the a.list of the affected center. As shown later, the functions that drive the operations check whether a center has such a pair. If so, the center is recognized as nonfunctional and the operational functions do nothing.

A more complex control function is KILL.CENTER:

```
(KILL.CENTER
  [LAMBDA (CENTER)
    (PROG ([MOD.LST (A.GETP# C.CENTERS (LIST CENTER (QUOTE LOCAL] STR)
          (MAPPRINQ ("The OK neighbors of " CENTER " are:  "))
          [PRIN1 (MAPSTRING (A.GETP# C.CENTERS
                                    (LIST CENTER (QUOTE OK.NBRS]
          (RPTQ 2 (TERPRI))
          [COND
            (MOD.LST (MAPC MOD.LST (FUNCTION (LAMBDA (ZZ)
                       (PROG (XX)
                             (MAPPRINQ ("Where should module " ZZ
                                                  " be moved?  "))
                             (SETQ XX (READ))
                             (A.ADDVAL# C.CENTERS
                                        (LIST XX (QUOTE LOCAL)) ZZ)
                             (A.PUT# C.CENTERS
                                     (LIST XX (QUOTE ROUTE) ZZ)
                                     XX]
          (A.PUT# C.CENTERS (LIST CENTER (QUOTE FAILED)) T)
          [REM.C.NBRS CENTER (A.GETP# C.CENTERS
                                    (LIST CENTER (QUOTE LINKS]
          (A.PUT# C.CENTERS (LIST CENTER (QUOTE LOCAL)))
          (SETQ STR (CONCAT "Center #" (MKSTRING CENTER)
                            " deactivated.  Modules "
                            (MAPSTRING MOD.LST T) " relocated."))
          (ADD.C.HISTORY (LIST STR])    .
```

This function assumes that the named center has failed and that the failure has been detected, perhaps by tests run by the center's neighbors. In any case, it allows the user to transfer all the local modules that were running in that center to new ones. It also removes the failed center from the OK.NBRS lists of its neighbors using REM.C.NBR:

```
(REM.C.NBRS
  [LAMBDA (CENTER NBR.LST)
    (MAPC NBR.LST (FUNCTION (LAMBDA (ZZ)
       (A.REMVAL# C.CENTERS (LIST ZZ (QUOTE OK.NBRS)) CENTER])
```

Another convenient function is MOVE.MOD, which transfers a named module to a new center. The center is left operable. This simulates the condition in which the module assignment functions cause a module to be moved to a new location in the network, perhaps to improve the load balance. This was the function used to generate Events 13 and 14 in Table 1.

```
(MOVE.MOD
  [LAMBDA (MOD NEW.LOC)
    (PROG ((CENTER (FIND.CENTER MOD)))
          (A.REMVAL# C.CENTERS (LIST CENTER (QUOTE LOCAL)) MOD)
          (A.REMPROP# C.CENTERS (LIST CENTER (QUOTE ROUTE) MOD))
          (A.REMPROP# C.CENTERS (LIST NEW.LOC (QUOTE ROUTE) MOD))
          (A.ADDVAL# C.CENTERS (LIST NEW.LOC (QUOTE LOCAL)) MOD)
          (A.ADDVAL# C.CENTERS (LIST NEW.LOC (QUOTE ROUTE))
                    (CONS MOD NEW.LOC))
          (COND (H.FLG
                 (ADD.H.STRING
                   (CONCAT "   << Module Shift >> :  Module #"
                           (MKSTRING MOD) " shifted to center #"
                           (MKSTRING NEW.LOC) " from #"
                           (MKSTRING CENTER) ".")))))])    .
```

MOVE.MOD removes the route information from the center that previously had the module and adds the appropriate information at the new location. Route information elsewhere in the network is not examined and will probably be faulty. If so, it will remain in error until an attempt is made to use it. The message-handling procedures are expected to work even in the presence of faulty route information and to correct that information when the existence of an error is detected.

This completes the description of the supporting, utility functions. We can now begin to consider the functions that actually drive the communication process.

b. High-Level Functions

The main top-level function used by the simulation for the basic communication operation is SEND.MSG. This function directs that a message be sent from the module named INIT.MOD to the module named TARGET.MOD:

```
(SEND.MSG
  [LAMBDA (INIT.MOD TARGET.MOD)
    (PROG ((CENTER (FIND.CENTER INIT.MOD))
           (SIZE (A.GETP C.CENTERS (QUOTE SIZE))) ID)
          (COND
            ((NULL SIZE)
              (MAPPRINQ ("Error.  No C.CENTER list available."
                         TERPRI))
              (RETURN NIL))
            ((NULL CENTER)
              (MAPPRINQ ("Error.  Module named " INIT.MOD
                         " cannot be located." TERPRI))
              (RETURN NIL)))
          (SETQ ID (GET.ID CENTER))
          (COND
            ((INIT.MSG INIT.MOD CENTER TARGET.MOD ID))
            (T (RETURN NIL)))
          (EXECUTE.SEND.MSG SIZE])     .
```

The function does some preliminary testing to confirm that the command is meaningful. In doing so it uses FIND.CENTER which looks through C.CENTERS to find the processor that contains INIT.MOD in its LOCAL list. (In an actual multiprocessor system, the module itself would initiate the message. The need for FIND.CENTER is a consequence of the simulation environment.)

```
(FIND.CENTER
  [LAMBDA (MOD)
    (PROG ((SIZE (A.GETP C.CENTERS (QUOTE SIZE))) (N 0))
      L   (COND ((LESSP N SIZE) (SETQ N (ADD1 N)))
                (T (RETURN NIL)))
          (COND ([MEMBER MOD (A.GETP# C.CENTERS (LIST N (QUOTE LOCAL]
                 (RETURN N)))
          (GO L])
```

SEND.MSG also calls GET.ID to create the unique ID that will be used for the message:

```
(GET.ID
  [LAMBDA (CENTER)
    (PROG [(N (A.GETP# C.CENTERS (LIST CENTER (QUOTE NEXT.ID]
          (A.PUT# C.CENTERS (LIST CENTER (QUOTE NEXT.ID)) (ADD1 N))
          (RETURN (MKATOM (CONCAT (MKSTRING CENTER) "-" (MKSTRING N])
```

This function forms the ID by concatenating the center's own number with "-" and the current value of the center's NEXT.ID. The value of NEXT.ID is increased by one.

SEND.MSG calls INIT.MSG, which actually creates the message and enters it into the system. The message is entered as a term in the list that is the value of the property named QUEUE in the a.list of the center.

```
(INIT.MSG
  [LAMBDA (INIT.MOD CENTER TARGET.MOD ID)
    (PROG [[MSG (LIST ID (CONS (QUOTE FROM) INIT.MOD)
                       (CONS (QUOTE AT) CENTER)
                       (CONS (QUOTE TO) TARGET.MOD)
                       (QUOTE (TYPE . ORIGINAL]
           (NEXT (A.GETP# C.CENTERS
                          (LIST CENTER (QUOTE ROUTE) TARGET.MOD]
          [COND
            ((AND TARGET.MOD NEXT)
              (A.PUT MSG (QUOTE MODE) (QUOTE DIRECTED))
              (A.PUT MSG (QUOTE ACK) T))
            (TARGET.MOD (A.PUT MSG (QUOTE MODE) (QUOTE SEARCH))
                        (A.PUT MSG (QUOTE ACK) T))
            (T (A.PUT MSG (QUOTE MODE) (QUOTE BDCST]
          (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE QUEUE)) MSG)
          [COND
            (H.FLG (ADD.C.HISTORY CENTER (COPY MSG) (QUOTE INIT]
          (RETURN MSG])    .
```

In creating the message, the function tries to find a route, setting NEXT to be the next center as indicated by the available route information. If no route information is available, NEXT nas the value NIL. The COND function sets the mode of the message according to the various conditions--i.e., whether a target module has been specified or not, and whether or not NEXT has a non-NIL value.

As its final action, SEND.MSG calls EXECUTE.SEND.MSG. This function does the work of transmitting the message. It does this by cycling through DEQUEUE.NTWK, L.DEQUEUE.NTWK, and PROC.NTWK, stopping only when it has gone through a complete cycle without a change. There is an additional wrinkle, however. After EXECUTE.SEND.MSG stops cycling, it calls TEST.NTWK, which can reinitiate the cycling process.

It guards against a premature exit from EXECUTE.SEND.MSG. The reason this final test is needed is discussed in section K.

EXECUTE.SEND.MSG is as follows:

```
(EXECUTE.SEND.MSG
  [LAMBDA (SIZE)
    (PROG (FLG)
      L   (COND ((DEQUEUE.NTWK SIZE) (SETQ FLG T)))
          (COND ((L.DEQUEUE.NTWK SIZE) (SETQ FLG T)))
          (COND ((PROC.NTWK SIZE) (SETQ FLG T)))
          (COND (FLG (SETQ FLG NIL) (GO L)))
          (COND ((TEST.NTWK SIZE) (GO L])    .
```

Each of the three functions called by EXECUTE.SEND.MSG cycles through the list of processors, calling the appropriate function to handle the list of messages that a single processor may have under the corresponding property name--i.e., QUEUE, L.QUEUE, and IN.PROCESS, respectively. The first is DEQUEUE.NTWK:

```
(DEQUEUE.NTWK
  [LAMBDA (SIZE)
    (PROG ((N 0) FLG)
          [COND ((NULL SIZE) (SETQ SIZE (A.GETP C.CENTERS (QUOTE SIZE]
      L   (COND ((LESSP N SIZE) (SETQ N (ADD1 N)))
                (T (RETURN FLG)))
          (COND ((DEQUEUE.LIST N) (SETQ FLG T)))
          (GO L])    ,
```

which calls the function DEQUEUE.LIST for each center.

The second is L.DEQUEUE.NTWK:

```
(L.DEQUEUE.NTWK
  [LAMBDA (SIZE)
    (PROG ((N 0) FLG)
          [COND ((NULL SIZE) (SETQ SIZE (A.GETP C.CENTERS (QUOTE SIZE]
      L   (COND ((LESSP N SIZE) (SETQ N (ADD1 N)))
                (T (RETURN FLG)))
          (COND ((L.DEQUEUE.LIST N)FLG T)))
          (GO L])    ,                    ,
```

which calls L.DEQUEUE.LIST for each center in turn.

The third function there is PROC.NTWK:

```
(PROC.NTWK
  [LAMBDA (SIZE)
    (PROG ((N 0) FLG)
          [COND ((NULL SIZE) (SETQ SIZE (A.GETP C.CENTERS (QUOTE SIZE]
      L   (COND ((LESSP N SIZE) (SETQ N (ADD1 N)))
                (T (RETURN FLG)))
          (COND ((PROC.LIST N) (SETQ FLG T)))
          (GO L])    ,
```

which calls PROC.LIST on each center.

DEQUEUE.LIST, which is called by DEQUEUE.NTWK on each center in turn, is as follows:

```
(DEQUEUE.LIST
  [LAMBDA (CENTER)
    (PROG ([LST (A.GETP# C.CENTERS (LIST CENTER (QUOTE QUEUE])
          (COND ((A.GETP# C.CENTERS (LIST CENTER (QUOTE FAILED)))
                 (A.PUT# C.CENTERS (LIST CENTER (QUOTE QUEUE)))
                 (RETURN NIL)))
          [COND ((NULL LST) (RETURN NIL))
                (T [MAPC LST (FUNCTION (LAMBDA (XX)
                     (COND ((DEQUEUE.MSG CENTER XX) (SETQ FLG T]
                   (A.PUT# C.CENTERS (LIST CENTER (QUOTE QUEUE]
          (RETURN T])    .
```

The main action of DEQUEUE.LIST is to apply DEQUEUE.MSG to each message in the list of messages it found under QUEUE for the center. As long as this list is not empty, DEQUEUE.LIST returns T, and so makes sure that EXECUTE.SEND.MSG will keep on going, at least for one more cycle. DEQUEUE.MSG, which is called on each message in the list in turn, is

```
(DEQUEUE.MSG
  [LAMBDA (CENTER MSG)
    (PROG [(LAST (A.GETP MSG (QUOTE LAST]
          [COND (LAST (A.ADDVAL# C.CENTERS (LIST LAST (QUOTE L.QUEUE))
                                   (LIST (CAR MSG)
                                         (CONS (QUOTE CENTER) CENTER)
                                         (QUOTE (TYPE . L-ACK]
          (COND (TRACE (PRINT.EVENT CENTER MSG)))
          (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE IN.PROCESS)) MSG])
```

DEQUEUE.MSG sends back a local acknowledgment to the center from which the message was received. When this has been done, it puts the message into the IN.PROCESS slot of the center for later processing.

L.DEQUEUE.NTWK, the second function in the cycle run by EXECUTE.SEND.MSG, calls L.DEQUEUE.LIST on each center in turn:

```
(L.DEQUEUE.LIST
  [LAMBDA (CENTER)
    (PROG ([LST (A.GETP# C.CENTERS (LIST CENTER (QUOTE L.QUEUE]
           ID.LST FLG)
          (COND ((A.GETP# C.CENTERS (LIST CENTER (QUOTE FAILED)))
                 (A.PUT# C.CENTERS (LIST CENTER (QUOTE L.QUEUE)))
                 (A.PUT# C.CENTERS (LIST CENTER (QUOTE BACKUP)))
                 (RETURN NIL)))
          (COND (LST [MAPC LST (FUNCTION (LAMBDA (ZZ)
                                          (L.DEQUEUE.MSG CENTER ZZ]
                     (A.PUT# C.CENTERS (LIST CENTER (QUOTE L.QUEUE)))
                     (SETQ FLG T)))
          [COND ((SETQ ID.LIST (TEST.L.QUEUE CENTER))
                 (MAPC ID.LIST (FUNCTION (LAMBDA (XX)
                                          (RECOVER.MSG CENTER XX]
          (A.REMPROP# C.CENTERS (LIST CENTER (QUOTE SENT)))
          (RETURN FLG])    .
```

The main action by L.DEQUEUE.LIST is the call of L.DEQUEUE.MSG on each message in the list retrieved by L.DEQUEUE.LIST. The function called is

```
(L.DEQUEUE.MSG
  [LAMBDA (CENTER MSG)
    (PROG [(TYPE (A.GETP MSG (QUOTE TYPE]
          (COND (L.TRACE (PRINT.EVENT CENTER MSG)))
          (COND ((EQUAL TYPE (QUOTE L-ACK)) (PROC.L.ACK CENTER MSG))
                ((EQUAL TYPE (QUOTE L-REF)) (PROC.L.REF CENTER MSG))
                ((EQUAL TYPE (QUOTE C-ACK)) (PROC.C.ACK CENTER MSG])
```

As has been stated, the local message system is intended to protect against certain failure situations. At the moment, we are examining the basic communication process, assuming no failures. Therefore, we shall not examine this function further at this time. We shall also defer further comment on the various types of local messages until we consider what is needed to cope successfully with failure and damage conditions.

The final function in the cycle executed by EXECUTE.SEND.MSG is PROC.NTWK, which calls PROC.LIST on each center:

```
(PROC.LIST
  [LAMBDA (CENTER)
    (PROG [(LST (A.GETP# C.CENTERS (LIST CENTER (QUOTE IN.PROCESS]
          (COND (LST [MAPC LST (FUNCTION (LAMBDA (XX)
                                   (PROC.MSG CENTER (COPY XX]
                     (A.PUT# C.CENTERS
                             (LIST CENTER (QUOTE IN.PROCESS)))
                     (RETURN T])
```

This function calls PROC.MSG on each of the messages in the list under IN.PROCESS for the given center. It returns T if the list is nonempty, again ensuring that EXECUTE.SEND.MSG will go through at least one more cycle. The function called on each message is

```
(PROC.MSG
  [LAMBDA (CENTER MSG)
    (PROG [(TYPE (A.GETP MSG (QUOTE TYPE))) (TO (A.GETP MSG (QUOTE TO]
          (COND
            ((NULL (PROC.ID CENTER MSG)))
            ([AND TO (MEMBER TO (A.GETP# C.CENTERS
                                         (LIST CENTER (QUOTE LOCAL]
              (ACCEPT.MSG CENTER MSG))
            ((EQUAL TYPE (QUOTE ORIGINAL)) (PROC.ORIG CENTER MSG))
            ((EQUAL TYPE (QUOTE C-ORIG)) (PROC.C.ORIG CENTER MSG))
            ((EQUAL TYPE (QUOTE ACK)) (PROC.ACK CENTER MSG))
            ((EQUAL TYPE (QUOTE C-ACK)) (PROC.C.ACK CENTER MSG))
            ((EQUAL TYPE (QUOTE CTR-STATUS))
              (PROC.CTR.STATUS CENTER MSG))
            ((EQUAL TYPE (QUOTE MSG-STATUS))
              (PROC.MSG.STATUS CENTER MSG])    .
```

PROC.MSG controls the processing of a message, branching according to the type. First, however, it tests whether the message has been seen before or is meant for a local module. The test for the newness of the message is done by PROC.ID:

```
(PROC.ID
  [LAMBDA (CENTER MSG)
    (PROG [(ID (CAR MSG)) (TYPE (A.GETP MSG (QUOTE TYPE]
          (COND
            ([MEMBER ID (A.GETP# C.CENTERS (LIST CENTER (QUOTE ACK]
              (RETURN NIL))
            ((EQUAL TYPE (QUOTE ACK))
              (A.REMVAL# C.CENTERS (LIST CENTER (QUOTE ORIG)) ID)
              (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE ACK)) ID)
              (RETURN T))
            [(MEMBER ID (A.GETP# C.CENTERS (LIST CENTER (QUOTE ORIG]
            (T (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE ORIG)) ID)
               (RETURN T)))
          (COND
            ((EQUAL (A.GETP MSG (QUOTE TYPE)) (QUOTE C-ORIG))
              (REFUSE.MSG CENTER MSG])    .
```

The main purpose of this function is to check the IDs. However, if the type is C-ORIG and its ID has been seen before, PROC.ID calls REFUSE.MSG. This function is discussed later in section k.

If the message has not been seen before, PROC.ID ascertains whether the target module of the message, if one exists, is in the list of LOCAL modules. If so, ACCEPT.MSG is called. This will be described in the next section; here we assume the target module is not local. The processing then branches according to the message type. Our assumption here is that the message is of type ORIGINAL, so that PROC.ORIG is called. In later sections we shall be considering the processing of each of the other types.

```
(PROC.ORIG
  [LAMBDA (CENTER MSG)
    (PROG [(MODE (A.GETP MSG (QUOTE MODE]
          (ADJUST.ROUTE CENTER MSG)
          (COND
            ((EQUAL MODE (QUOTE BDCST)) (PROC.BDCST CENTER MSG))
            ((EQUAL MODE (QUOTE SEARCH)) (PROC.SEARCH CENTER MSG))
            ((EQUAL MODE (QUOTE DIRECTED)) (PROC.DIRECTED CENTER MSG))
            ((EQUAL MODE (QUOTE DEFAULT)) (PROC.DEFAULT CENTER MSG])
```

This function first calls ADJUST.ROUTE to enter the backward route information implied by the message--i.e., the route to the module that was its original source.

```
(ADJUST.ROUTE
  [LAMBDA (CENTER MSG)
    (PROG [(LAST (A.GETP MSG (QUOTE LAST)))
           (MOD (A.GETP MSG (QUOTE FROM]
          (COND ((AND LAST MOD)
                 (A.PUT# C.CENTERS (LIST CENTER (QUOTE ROUTE) MOD)
                         LAST])
```

PROC.ORIG then branches according to the mode specified in the message. Note that no tests are made of the suitability of the existing mode.

Two cases arise. The message may be in the BDCST mode. If so, it will be propagated through the network without any change of mode. Or the message may name a target module when it can be in SEARCH, DIRECTED, or DEFAULT mode with some changes among the three being possible. We consider these two cases separately.

c. Broadcast Messages

For a message of ORIGINAL type in BDCST mode, no mode change is allowed and no acknowledgment expected (other than the local acknowledgments discussed later). The intention is to flood the network with the message. It is, therefore, the simplest procedure. Duplications must be avoided lest the message propagate indefinitely, but the ID of the message has already been checked in PROC.MSG before PROC.ORIG is reached. PROC.BDCST, therefore, is

```
(PROC.BDCST
  [LAMBDA (CENTER MSG)
    (PROG [(LAST (A.GETP MSG (QUOTE LAST]
          (SETQ MSG (COPY MSG))
          (COND ((SETQ LST (NEXT.LIST CENTER LAST))
                 (A.PUT MSG (QUOTE LAST) CENTER)
                 (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE SENT))
                            (CONS ID LST))
                 [MAPC LST (FUNCTION (LAMBDA (ZZ)
                   (A.ADDVAL# C.CENTERS (LIST ZZ (QUOTE QUEUE)) MSG]
                 (COND (H.FLG (ADD.C.HISTORY CENTER (COPY MSG)
                                             (QUOTE BDCST) LST])
```

which puts copies of the message in the QUEUE of each center into the list developed by NEXT.LIST.

It may be wondered why the message is copied before it is relayed. This is done because many of the INTERLISP operations use pointers to lists, rather than the actual lists. Therefore, if a message is passed to several centers without using the COPY function,

what is transferred may actually be multiple pointers to a single message. If, say, the mode of the message is then changed at one center, the messages at all centers are also changed. To avoid this phenomenon, which is both unwanted and unrealistic for the multiprocessor environment, we make copies whenever a message is passed around.

The function NEXT.LIST that develops the list of neighbors to which the message is sent is

```
(NEXT.LIST
  [LAMBDA (CENTER LAST)
    (PROG [(LST (COPY (A.GETP# C.CENTERS (LIST CENTER (QUOTE OK.NBRS]
          (COND ((NULL LAST) (RETURN LST))
                ((OR (NULL LST) (EQUAL LST (LIST LAST))) (RETURN NIL))
                (T (RETURN (REMOVE LAST LST])
```

This uses the list of OK.NBRS of the center, but removes the neighbor, if any, that is specified as the value of LAST when it is called. We avoid propagating the message back along the route by which it arrived at a center. It would not be a major fault if we allowed a backward propagation since the check of a message's ID would prevent the propagation from continuing indefinitely. But it would be an obvious inefficiency that can be avoided easily.

d. Module-to-Module Messages

If a target module is named in SEND.MSG, the message is initiated and propagated as an ORIGINAL type of message in one of the three modes: SEARCH, DIRECTED, and DEFAULT. It is initiated by INIT.MSG in either the SEARCH or DIRECTED mode, depending on whether route information to the target module is absent or present in the center. Subsequently the SEARCH mode may be changed to DIRECTED if route information is discovered, or DIRECTED changed to DEFAULT if an error is discovered in the route information being used. PROC.ORIG branches according to the mode of the message to one of PROC.SEARCH, PROC.DIRECTED, or PROC.DEFAULT. PROC.SEARCH, however, will call PROC.DIRECTED if route information is available, and PROC.DIRECTED will call PROC.DEFAULT, if necessary.

```
(PROC.SEARCH
  [LAMBDA (CENTER MSG)
    (PROG ([ROUTE (TEST.ROUTE CENTER (A.GETP MSG (QUOTE TO]
           (LAST (A.GETP MSG (QUOTE LAST))) LST)
          (SETQ MSG (COPY MSG))
          (COND (ROUTE (A.PUT MSG (QUOTE MODE) (QUOTE DIRECTED))
                       (PROC.DIRECTED CENTER MSG))
                [(SETQ LST (NEXT.LIST CENTER LAST))
                  (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE SENT))
                             (CONS ID LST))
                  (A.ADDVAL# C.CENTERS
                             (LIST CENTER (QUOTE BACKUP)) MSG)
                  (A.PUT MSG (QUOTE LAST) CENTER)
                  [MAPC LST (FUNCTION (LAMBDA (ZZ)
                    (A.ADDVAL# C.CENTERS (LIST ZZ (QUOTE QUEUE)) MSG]
                  (COND (H.FLG (ADD.C.HISTORY CENTER (COPY MSG)
                                              (QUOTE SEARCH) LST]
                (T (WARN.ORIG CENTER MSG])    ,
```

where NEXT.LIST, which generates the list of neighbors to which the message will be sent, is the function described before.

The default situation, when there is no route information and nobody to relay the message to as a continuation of the SEARCH mode, causes WARN.ORIG to be called. This returns a MSG-STATUS type of message to the originator of the message, warning that the message may be blocked. This function is discussed in Section I.

```
(PROC.DIRECTED
  [LAMBDA (CENTER MSG)
    (PROG [(ROUTE (TEST.ROUTE CENTER (A.GETP MSG (QUOTE TO]
          (SETQ MSG (COPY MSG))
          (COND
            [ROUTE (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE SENT))
                               (LIST (CAR MSG) ROUTE))
                   (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE BACKUP))
                               MSG)
                   (A.PUT MSG (QUOTE LAST) CENTER)
                   (A.ADDVAL# C.CENTERS (LIST ROUTE (QUOTE QUEUE))
                               MSG)
                   (COND
                     (H.FLG (ADD.C.HISTORY CENTER (COPY MSG)
                                           (QUOTE DIRECTED) ROUTE]
            (T (WARN.ORIG CENTER MSG)
               (A.PUT MSG (QUOTE MODE) (QUOTE DEFAULT))
               (PROC.DEFAULT CENTER MSG])    ,
```

```
(PROC.DEFAULT
  [LAMBDA (CENTER MSG)
    (PROG ((LAST (A.GETP MSG (QUOTE LAST))) LST)
          (SETQ MSG (COPY MSG))
          (COND
            [(SETQ LST (NEXT.LIST CENTER LAST))
              (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE SENT))
                         (CONS ID LST))
              (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE BACKUP)) MSG)
              (A.PUT MSG (QUOTE LAST) CENTER)
              [MAPC LST (FUNCTION (LAMBDA (ZZ)
                (A.ADDVAL# C.CENTERS (LIST ZZ (QUOTE QUEUE)) MSG]
              (COND
                (H.FLG (ADD.C.HISTORY CENTER (COPY MSG)
                                      (QUOTE DEFAULT) LST]
            (T (WARN.ORIG CENTER MSG])    .
```

The entries in the lists that are the values of SENT and BACKUP are used for detecting and recovering from failed centers. We shall discuss them later.

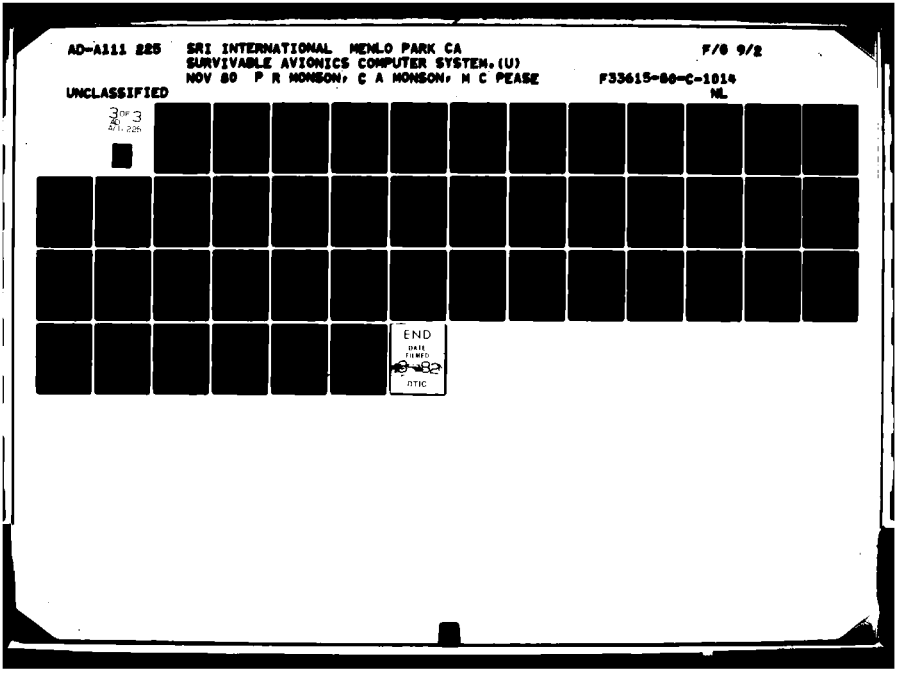
As in PROC.SEARCH, PROC.DEFAULT will not be able to relay the message anywhere if NEXT.LIST returns NIL. This will be true if the only operable neighbor is the one from which the center received the message. In this case, PROC.DEFAULT calls WARN.ORIG to warn the originator of the message that there may be trouble. As stated, this function will be discussed later.

The test of the route information that controls the various mode shifts is made by TEST.ROUTE:

```
(TEST.ROUTE
  [LAMBDA (CENTER MOD)
    (PROG [(ROUTE (A.GETP# C.CENTERS (LIST CENTER (QUOTE ROUTE) MOD)))
           (OK.NBRS (A.GETP# C.CENTERS (LIST CENTER (QUOTE OK.NBRS]
          (COND
            ((NULL ROUTE) (RETURN NIL))
            ((NULL OK.NBRS) (RETURN NIL))
            ((MEMBER ROUTE OK.NBRS) (RETURN ROUTE))
            (T (RETURN NIL])    .
```

This function checks whether acceptable route information to the module MOD is available. If no route information is available or if the specified route is not among the center's OK.NBRS, TEST.ROUTE returns NIL.

These, then, are the functions that cause the usual messages to be propagated through the network, using route information if available, otherwise employing a broadcast mode of transmission. If


AD-A111 825
SRI INTERNATIONAL MENLO PARK CA
SURVIVABLE AVIONICS COMPUTER SYSTEM.(U)
NOV 80 P R MONSON, C A MONSON, M C PEASE
F/G 9/2
F33615-80-C-1014
UNCLASSIFIED
NL
END
DATE FILMED
DTIC

the route information is incorrect, it will still be followed as long as the existence of the error is not apparent. If, for example, a module has been moved, a message to that module may follow a previously set route to its old location. At that point the route information fails and the message shifts to the DEFAULT mode, searching for the new location of the module. This does not necessarily lead to the shortest possible route, but it will lead to a successful route whenever one exists.

e. Message Acceptance.

When a message specifies a target module and PROC.MSG recognizes that that target module is a local one, propagation of that message from that center is stopped and ACCEPT.MSG is called.

```
(ACCEPT.MSG
  [LAMBDA (CENTER MSG)
    (PROG [(ID (CAR MSG)) (TYPE (A.GETP MSG (QUOTE TYPE]
          (ADJUST.ROUTE CENTER MSG)
          (COND
            ((EQUAL TYPE (QUOTE ORIGINAL))
              [MAPPRINQ ("Msg #" ID " received by "
                         (A.GETP MSG (QUOTE  ) " at " CENTER "."
                         TERPRI "Acknowledgment sent."
                         (RPTQ 2 (TERPRI]
              (A.REMPROP# C.CENTERS (LIST CENTER (QUOTE BACKUP) ID))
              (COND
                ((A.GETP MSG (QUOTE ACK)) (INIT.ACK CENTER MSG))))
            ((EQUAL TYPE (QUOTE ACK))
              (MAPPRINQ ("Acknowledgment of msg #" ID " received by "
                         (A.GETP MSG (QUOTE TO)) " at " CENTER "."
                         TERPRI
                         "Processing of this message complete."
                         TERPRI))
              (A.REMPROP# C.CENTERS
                          (LIST CENTER (QUOTE BACKUP) ID)))
            ((EQUAL TYPE (QUOTE MSG-STATUS))
              (MAPPRINQ ("Warning received from center #"
                         (A.GETP MSG (QUOTE AT)) " that msg #"
                         (A.GETP MSG (QUOTE MSG-ID))
                         " may be blocked." TERPRI])    .
```

This function does not generate an acknowledgment to a C-ORIG type of message since this type does not specify a target module. The response to a C-ORIG message is discussed later.

ACCEPT.MSG first determines whether the message type is ORIGINAL, ACK, or MSG-STATUS. It prints out a statement of the situation that is convenient for the simulation. If the message was an acknowledgment or a MSG-STATUS type, nothing further is done except to discard residual information no longer needed. If the message was of the type ORIGINAL, ACCEPT.MSG determines if the property ACK has a non-NIL value. If so, INIT.ACK is called to create an acknowledgment:

```
(INIT.ACK
  [LAMBDA (CENTER MSG)
    (PROG [(ACK (LIST (CAR MSG)
                      (CONS (QUOTE FROM) (A.GETP MSG (QUOTE TO)))
                      (CONS (QUOTE AT) CENTER)
                      (CONS (QUOTE TO) (A.GETP MSG (QUOTE FROM)))
                      (QUOTE (TYPE . ACK))
                      (QUOTE (MODE . DIRECTED]
          (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE QUEUE)) ACK)
          (COND
            (H.FLG (ADD.C.HISTORY CENTER (COPY ACK) (QUOTE ACK-INIT])
```

This formats the acknowledgment properly and adds it to the center's QUEUE, where it will be handled the next time DEQUEUE.LIST is called at the center. Note that the acknowledgment has the same ID as the message being acknowledged. The acknowledgment is launched in the DIRECTED mode. Since the message has just been received, the center must have route information leading to the source of the message.

f. Propagation of Acknowledgments

An acknowledgment is handled the same as any other message until PROC.MSG examines it. At that point, its type is recognized and PROC.ACK is called instead of PROC.ORIG.

```
(PROC.ACK
  [LAMBDA (CENTER MSG)
    (PROG [(MODE (A.GETP MSG (QUOTE MODE]
          (ADJUST.ROUTE CENTER MSG)
          (COND
            ((EQUAL MODE (QUOTE DIRECTED)) (PROC.ACK.DIR CENTER MSG))
            ((EQUAL MODE (QUOTE DEFAULT))
              (PROC.ACK.DEFAULT CENTER MSG])    .
```

Like PROC.ORIG, PROC.ACK branches according to the mode after updating the route information with ADJUST.ROUTE. It calls either PROC.ACK.DIR or PROC.ACK.DEFAULT:

```
(PROC.ACK.DIR
  [LAMBDA (CENTER MSG)
    (PROG [(ROUTE (TEST.ROUTE CENTER (A.GETP MSG (QUOTE TO]
          (SETQ MSG (COPY MSG))
          (COND [ROUTE (A.PUT MSG (QUOTE LAST) CENTER)
                       (A.ADDVAL# C.CENTERS
                                  (LIST ROUTE (QUOTE QUEUE)) MSG)
                       (COND (H.FLG (ADD.C.HISTORY CENTER (COPY MSG)
                                                   (QUOTE ACK-DIR) ROUTE]
                (T (A.PUT MSG (QUOTE MODE) (QUOTE DEFAULT))
                   (PROC.ACK.DEFAULT CENTER MSG])    ,

(PROC.ACK.DEFAULT
  [LAMBDA (CENTER MSG)
    (PROG ((LAST (A.GETP MSG (QUOTE LAST))) LST)
          (SETQ MSG (COPY MSG))
          [COND ((SETQ LST (NEXT.LIST CENTER LAST))
                 (A.PUT MSG (QUOTE LAST) CENTER)
                 [MAPC LST (FUNCTION (LAMBDA (ZZ)
                   (A.ADDVAL# C.CENTERS (LIST ZZ (QUOTE QUEUE)) MSG]
                 (COND (H.FLG (ADD.C.HISTORY CENTER (COPY MSG)
                                             (QUOTE ACK.DEFAULT) LST]])    .
```

Like the interaction between PROC.DIRECTED and PROC.DEFAULT, PROC.ACK.DIRECTED can pass the message over to PROC.ACK.DEFAULT if necessary. Indeed, these two functions are essentially stripped-down versions of PROC.DIRECTED and PROC.DEFAULT, respectively. The differences result from the fact that we do not need to take precautions to ensure that an acknowledgment reaches its destination. At worst, a failure may simply cause the original message to be repeated.

The fact that an acknowledgment has indeed reached its destination is recognized by PROC.MSG, which calls ACCEPT.MSG, discussed before.

g. Detection of Failed Centers

The material discussed so far describes the basic message-handling procedures. We now consider the processes that protect against malfunctions. The first requirement is to test for centers that may have failed.

As stated, a failed center is recognized by noticing that not all the expected local acknowledgments have been received. To do this, the list of centers to which a message has been sent is recorded under the property name SENT. The value of SENT is a list of lists.

Each sublist is headed by the ID of the message involved and its body is the list of centers. Each time a local acknowledgment is received, the appropriate entry in the list that is the value of SENT is removed. We expect the SENT list to be wiped out by the time all local acknowledgments have been processed. If not, the remaining entries in it are presumed to be faulty and appropriate action is taken.

While waiting for verification that the message has been received and acted upon, we need to retain information that will be needed in case of trouble. The easiest way to do this is to keep a copy of the message. This is done under the property name BACKUP.

We have already seen the creation of SENT entries and the backup copy of the message in PROC.SEARCH, PROC.DIRECTED, and PROC.DEFAULT. We have also seen that DEQUEUE.MSG generates the necessary local acknowledgments as the message is dequeued, provided the center does not have a non-NIL value of the property FAILED. Finally, we have seen that local acknowledgments are recognized by L.DEQUEUE.MSG. The latter calls PROC.L.ACK for each local acknowledgment.

```
(PROC.L.ACK
  [LAMBDA (CENTER MSG)
    (PROG [[SENT.LIST (A.GETP# C.CENTERS
                              (LIST CENTER (QUOTE SENT) (CAR MSG]
           [TEST.LIST (A.GETP# C.CENTERS
                              (LIST CENTER (QUOTE TEST) (CAR MSG]
           (ID (CAR MSG))
           (SOURCE (A.GETP MSG (QUOTE CENTER]
          (COND ((NULL SENT.LIST))
                ((EQUAL SENT.LIST (LIST SOURCE))
                  (A.REMPROP# C.CENTERS (LIST CENTER (QUOTE SENT) ID))
                  (A.REMPROP# C.CENTERS
                              (LIST CENTER (QUOTE BACKUP) ID)))
                (T (A.REMVAL# C.CENTERS (LIST CENTER (QUOTE SENT) ID)
                              SOURCE)))
          (COND ((NULL TEST.LIST))
                ((EQUAL TEST.LIST (LIST SOURCE))
                  (A.REMPROP# C.CENTERS
                              (LIST CENTER (QUOTE TEST) ID)))
                (T (A.REMVAL# C.CENTERS (LIST CENTER (QUOTE TEST) ID)
                              SOURCE]))    .
```

The immediate effect is to modify the sublist with the message's ID in the list that is the value of SENT for the center. If the effect is to annihilate that sublist, the center knows there is no immediate problem. To reduce the unnecessary storage, it eliminates the backup copy of the message .

The final condition has to do with the processing of C-ORIG messages and is discussed later.

The decision on whether or not there is trouble is made at the end of the call of L.DEQUEUE.LIST on a center. By this time all local messages to the center have been processed. The SENT list is examined by TEST.L.QUEUE.

```
(TEST.L.QUEUE
  [LAMBDA (CENTER)
    (PROG ([SENT (A.GETP# C.CENTERS (LIST CENTER (QUOTE SENT] LST)
          (COND ((SETQ LST (CONSOLIDATE.LIST SENT))
                 [MAPC LST (FUNCTION (LAMBDA (XX)
                                       (NOTE.FAILURE CENTER XX]
                 (RETURN (CAR.LIST SENT)))
                (T (COND (L.TRACE
                          (MAPPRINQ ("At center #" CENTER ":"
                                     (TAB 10)
                                     "All required L.ACKs received."
                                     TERPRI])    .
```

CONSOLIDATE.LIST is used on the value of SENT because the latter has a complex structure. As has been shown, the value of SENT is a list of lists, each headed by a message ID and followed by a list of centers. While normally there will be only one sublist, we do not want to restrict the simulation. CONSOLIDATE.LIST builds up LST by first setting it equal to the CDAR of LIST.LIST. Subsequent sublists in LIST.LIST are examined and their terms added to LST only if they do not duplicate any existing entry. CONSOLIDATE.LIST returns a simple list of centers without duplications.

```
(CONSOLIDATE.LIST
  [LAMBDA (LIST.LIST)
    (PROG (LST)
          [MAPC LIST.LIST (FUNCTION (LAMBDA (SUBLST)
             (COND ((NULL LST) (SETQ LST (CDR SUBLST)))
                   (T (MAPC (CDR SUBLST) (FUNCTION (LAMBDA (XX)
                         (COND ((NOT (MEMBER XX LST))
                                (SETQ LST (CONS XX LST]
          (RETURN LST])
```

If TEST.L.QUEUE finds SENT to be empty, there is no problem. The center has received all the required local acknowledgments.

If SENT is not empty, it is a list of neighbors that are presumed to be faulty. TEST.L.QUEUE calls NOTE.FAILURE on each distinct center remaining in SENT and generates a CTR-STATUS message broadcasting the observation.

```
(NOTE.FAILURE
  [LAMBDA (CENTER FAILED.CTR)
    (COND
      ([MEMBER FAILED.CTR (A.GETP# C.CENTERS
                                   (LIST CENTER (QUOTE OK.NBRS]
        (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE QUEUE))
                   (LIST (GET.ID CENTER)
                         (CONS (QUOTE AT) CENTER)
                         (QUOTE (TYPE . CTR-STATUS))
                         (QUOTE (MODE . BDCST))
                         (CONS (QUOTE CENTER) FAILED.CTR)
                         (QUOTE (STATUS . FAILED])    .
```

Finally, TEST.L.QUEUE returns a list of the IDs of the message(s) that may have been affected by the failure and for which special action may be required. The actions taken to recover operations are discussed in Subsection i. The list of the IDs of the possibly affected messages are obtained from the remaining value of SENT by using CAR.LIST to develop a list of the lead terms of the list of lists that is SENT:

```
(CAR.LIST
  [LAMBDA (LST)
    (COND ((LISTP LST) (MAPCAR LST (FUNCTION (LAMBDA (ZZ) (CAR ZZ])
```

h. CTR-STATUS-Type Messages

The CTR-STATUS type of message is currently used only to broadcast the information that a given center has failed. We might also expect to use this type when a center recovers from a failure, or whenever its capability or availability changes for any reason. These additional applications have not been included in the simulation since doing so would not introduce anything new. The use of CTR-STATUS for reporting failures is a sufficient context in which to examine the type.

PROC.MSG calls PROC.CTR.STATUS when it discovers the type. PROC.CTR.STATUS in turn calls PROC.CTR.FAILED. This is done to leave room for branching.

```
(PROC.CTR.STATUS
  [LAMBDA (CENTER MSG)
    (PROG NIL
          (COND ((EQUAL (A.GETP MSG (QUOTE STATUS)) (QUOTE FAILED))
                 (PROC.CTR.FAILED CENTER MSG])
```

PROC.CTR.FAILED causes the center receiving the message to examine the list of OK.NBRS. If the failed center is contained in that list, the list is modified to exclude it. The message is then broadcast to all remaining neighbors.

```
(PROC.CTR.FAILED
  [LAMBDA (CENTER MSG)
    (PROG ((LAST (A.GETP MSG (QUOTE LAST)))
           (FAILED (A.GETP MSG (QUOTE CENTER))) LST)
          (A.REMVAL# C.CENTERS (LIST CENTER (QUOTE OK.NBRS)) FAILED)
          (COND ((SETQ LST (NEXT.LIST CENTER LAST))
                 (SETQ MSG (COPY MSG))
                 (A.PUT MSG (QUOTE LAST) CENTER)
                 (MAPC LST (FUNCTION (LAMBDA (ZZ)
                   (A.ADDVAL# C.CENTERS (LIST ZZ (QUOTE QUEUE)) MSG])
```

i. Unblocking Messages

As described in Section g, (Detection of Failed Centers), TEST.L.QUEUE is called by L.DEQUEUE.LIST to determine whether all expected local acknowledgments have been received. If not, the center(s) failing to send local acknowledgments are noted and CTR-STATUS-type messages initiated by NOTE.FAILURE. TEST.L.QUEUE returns a a list of the IDs of messages that may have been affected.

This still leaves the problem of what to do about the message(s) for which the required set of local acknowledgments has not been received. The immediate need is to recover communication, if possible, from the point at which the failure was detected. This is done by calling RECOVER.MSG on each ID in the list of potentially affected messages returned by TEST.L.QUEUE.

```
(RECOVER.MSG
  [LAMBDA (CENTER ID)
    (PROG [(MSG (CONS ID (A.GETP# C.CENTERS
                                  (LIST CENTER (QUOTE BACKUP) ID]
          (A.REMPROP# C.CENTERS (LIST CENTER (QUOTE BACKUP) ID))
          (COND ((NULL MSG) (RETURN NIL))
                ((NOT (EQUAL (A.GETP MSG (QUOTE TYPE))
                             (QUOTE ORIGINAL)))
                  (RETURN NIL)))
          (COND
            ((EQUAL (A.GETP MSG (QUOTE MODE)) (QUOTE DIRECTED))
              (A.PUT MSG (QUOTE MODE) (QUOTE DEFAULT))
              (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE QUEUE)) MSG)))
          (WARN.ORIG CENTER (CONS ID MSG])
```

This function is intended to take whatever local action may be possible and necessary to recover a transmission that may have been blocked by failure. If appropriate, it also warns the message's originator of possible trouble. It uses the backup copy of the message. First, it examines the type of the message. If the type is not ORIGINAL, nothing more is needed. If the type is ORIGINAL and the mode is SEARCH, again no action is needed. This is because, if the mode is SEARCH, it has been so since the message was originated. If there is a path to the target module, either it will have been found in the SEARCH mode or the propagation along some other path will have been changed to DIRECTED (and then perhaps to DEFAULT) and subsequently blocked. In that case, the blockage of the DIRECTED or DEFAULT mode will call RECOVER.MSG under more sensitive conditions.

If the type is ORIGINAL and the mode DIRECTED, the mode is changed to DEFAULT and the message is reprocessed. This does not eliminate the need for a warning message, but does increase the chance that the message may go through without being reinitiated. Finally, if the type is ORIGINAL and the mode DEFAULT, there is no way to recover transmission locally. All that can be done is to advise the originating module. The warning message, in any of the cases requiring it, is created by WARN.ORIG:

```
(WARN.ORIG
  [LAMBDA (CENTER MSG)
    (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE QUEUE))
               (LIST (GET.ID CENTER)
                     (CONS (QUOTE AT) CENTER)
                     (CONS (QUOTE TO) (A.GETP MSG (QUOTE FROM)))
                     (CONS (QUOTE MSG-ID) (CAR MSG))
                     (QUOTE (TYPE . MSG-STATUS))
                     (QUOTE (MODE . DIRECTED))
                     (QUOTE (STATUS . BLOCKED])    .
```

WARN.ORIG sets its own ID, using GET.ID, so as to avoid interference with the message about which the warning is being sent. The warning message is launched in the DIRECTED mode, but can shift to the DEFAULT mode if necessary.

j. MSG-STATUS-Type Messages

A MSG-STATUS type of message will always be initiated in the DIRECTED mode, but may need to be changed to the DEFAULT mode if it cannot backtrack the message whose status is being reported. Therefore, when PROC.MSG recognizes that the message is of the type MSG-STATUS, it calls PROC.MSG.STATUS, which branches on the mode:

```
(PROC.MSG.STATUS
  [LAMBDA (CENTER MSG)
    (PROG [(MODE (A.GETP MSG (QUOTE MODE]
          (COND ((EQUAL MODE (QUOTE DIRECTED))
                 (PROC.MSG.STATUS.DIR CENTER MSG))
                ((EQUAL MODE (QUOTE DEFAULT))
                 (PROC.MSG.STATUS.DEFAULT CENTER MSG])
```

This function calls either of the following two functions:

```
(PROC.MSG.STATUS.DIR
  [LAMBDA (CENTER MSG)
    (PROG [(ROUTE (TEST.ROUTE CENTER (A.GETP MSG (QUOTE TO]
          (SETQ MSG (COPY MSG))
          (COND
            [ROUTE (A.PUT MSG (QUOTE LAST) CENTER)
                   (A.ADDVAL# C.CENTERS
                              (LIST ROUTE (QUOTE QUEUE)) MSG)
                   (COND
                     (H.FLG (ADD.C.HISTORY CENTER (COPY MSG)
                                           (QUOTE MSG-STATUS-DIR)
                                           ROUTE]
            (T (A.PUT MSG (QUOTE MODE) (QUOTE DEFAULT))
               (PROC.MSG.STATUS.DEFAULT CENTER MSG])    ,
```

or

```
(PROC.MSG.STATUS.DEFAULT
  [LAMBDA (CENTER MSG)
    (PROG ((LAST (A.GETP MSG (QUOTE LAST))) LST)
          (SETQ MSG (COPY MSG))
          (COND
            ((SETQ LST (NEXT.LIST CENTER LAST))
              (A.PUT MSG (QUOTE LAST) CENTER)
              [MAPC LST (FUNCTION (LAMBDA (ZZ)
                 (A.ADDVAL# C.CENTERS (LIST ZZ (QUOTE QUEUE)) MSG]
              (COND
                (H.FLG (ADD.C.HISTORY CENTER (COPY MSG)
                                      (QUOTE MSG-STATUS-DEF) LST])    .
```

As with the corresponding functions for other types, the former can call the latter if route information is not available.

k. C-ORIG-Type Messages

The purpose of a C-ORIG type of message is generally to enable a center to determine what other centers are operative. Presumably it would be used after massive damage to assess what facilities remain operative. This intention is reflected in the name of the top-level function that creates and transmits the type, ASSESS.NTWK:

```
(ASSESS.NTWK
  [LAMBDA (CENTER)
    (PROG ((ID (GET.ID CENTER)) MSG)
          [SETQ MSG (LIST ID (CONS (QUOTE AT) CENTER)
                          (QUOTE (TYPE . C-ORIG))
                          (QUOTE (MODE . BDCST]
          (A.ADDVAL# C.CENTERS (LIST CENTER (QUOTE QUEUE)) MSG)
          [COND (H.FLG (ADD.C.HISTORY CENTER (COPY MSG)
                                      (QUOTE C-ORIG-INIT]
          (EXECUTE.SEND.MSG (A.GETP C.CENTERS (QUOTE SIZE)))
          (STRIP.NTWK])    .
```

This function creates the proper format, enters it into the QUEUE of the center, and then calls EXECUTE.SEND.MSG as did SEND.MSG. Finally it calls STRIP.NTWK to eliminate execess material in the representation of the network. STRIP.NTWK is

```
(STRIP.NTWK
  [LAMBDA NIL
    (PROG ((SIZE (A.GETP C.CENTERS (QUOTE SIZE))) (N 0))
      L   (COND ((LESSP N SIZE) (SETQ N (ADD1 N)))
                (T (RETURN NIL)))
          (STRIP.CENTER N)
          (GO L])    ,
```

which calls STRIP.CENTER on each center in turn:

```
(STRIP.CENTER
  [LAMBDA (CENTER)
    (PROG ((C.LIST (A.GETP C.CENTERS CENTER)) NEW.LIST)
          [SETQ NEW.LIST (LIST (CONS (QUOTE LINKS)
                                     (COPY (A.GETP C.LIST
                                                   (QUOTE LINKS]
```

```
[MAPC C.LIST (FUNCTION (LAMBDA (ZZ)
          (COND ((EQUAL (CAR ZZ) (QUOTE LINKS)))
                ((OR (EQUAL (CAR ZZ) (QUOTE OK.NBRS))
                     (EQUAL (CAR ZZ) (QUOTE LOCAL))
                     (EQUAL (CAR ZZ) (QUOTE NEXT.ID))
                     (EQUAL (CAR ZZ) (QUOTE ROUTE))
                     (AND (EQUAL (CAR ZZ) (QUOTE FAILED))
                          (CDR ZZ)))
                 (SETQ NEW.LIST (CONS (COPY ZZ) NEW.LIST]
(SETQ NEW.LIST (APPEND (COPY (QUOTE ((L.QUEUE) (QUEUE)
                                     (IN.PROCESS)))
                       NEW.LIST))
(A.PUT C.CENTERS CENTER (REVERSE NEW.LIST])    .
```

When DEQUEUE.MSG is called on the C-ORIG message, it transfers the message to the IN.PROCESS list and generates a local acknowledgment that is returned to the immediate source of the message. This is exactly the same as with ORIGINAL messages.

When PROC.MSG is called on the message in the IN.PROCESS list, provided its ID has not been seen before, the type is recognized and the message is passed to PROC.C.ORIG:

```
(PROC.C.ORIG
  [LAMBDA (CENTER MSG)
    (PROG ((LAST (A.GETP MSG (QUOTE LAST))) (ID (CAR MSG)) LST)
          (SETQ MSG (COPY MSG))
          (COND
            (LAST (A.PUT# C.CENTERS (LIST CENTER (QUOTE SOURCE) ID)
                          LAST))
            (T (A.PUT# C.CENTERS (LIST CENTER (QUOTE SOURCE) ID) T)))
          (COND
            [(SETQ LST (NEXT.LIST CENTER LAST))
              (A.PUT MSG (QUOTE LAST) CENTER)
              (A.ADDPROP# C.CENTERS (LIST CENTER (QUOTE SENT) ID)
                          (COPY LST))
              (A.ADDPROP# C.CENTERS (LIST CENTER (QUOTE WAIT) ID)
                          (COPY LST))
              [MAPC LST (FUNCTION (LAMBDA (ZZ)
                 (A.ADDVAL# C.CENTERS (LIST ZZ (QUOTE QUEUE)) MSG]
              (COND
                (H.FLG (ADD.C.HISTORY CENTER (COPY MSG)
                                      (QUOTE C-ORIG) LST]
            (T (INIT.C.ACK CENTER ID])    .
```

This function does certain things that the other process functions do not have to do. First, under the property name SOURCE,

it stores a record of where the message came from. Note that this is not the original source, but the center that put the message into the receiving center's queue. The purpose is to retain knowledge of which neighbor should receive the acknowledgment when the time comes. Since the C-ACK acknowledgment may not be returned until many cycles later, the message itself could have long ceased to exist. If there is no value to LAST in the message, SOURCE is given the value T, indicating that that center is the original source of the message.

PROC.C.ORIG also stores the list of centers to which copies were sent, not only under SENT as before, but also under the property name WAIT. The value under SENT will be used to check off the local acknowledgments as before. The value under WAIT will be used to check off the return of either a local refusal (L-REF type) or an acknowledgment of the C-ACK type. The remaining value under WAIT at various stages in the processing will be used to determine whether the center will itself initiate a C-ACK return to the center from which it received the C-ORIG-type message.

We now consider the various ways information is returned to a center.

k1. Refused Messages:

If the ID of a C-ORIG-type message has been seen before, the message is not simply ignored, as was the case with ORIG type messages. Instead, PROC.ID calls REFUSE.MSG, which generates an L-REF-type local message. This is needed so that the source will know that it should not wait for an acknowledgment from that center.

```
(REFUSE.MSG
  [LAMBDA (CENTER MSG)
    (PROG [(TYPE (A.GETP MSG (QUOTE TYPE)))
           (LAST (A.GETP MSG (QUOTE LAST]
          (COND (LAST (A.ADDVAL# C.CENTERS (LIST LAST (QUOTE L.QUEUE))
                                   (LIST (CAR MSG)
                                         (CONS (QUOTE CENTER) CENTER)
                                         (QUOTE (TYPE . L-REF])
```

When the L.QUEUE of the source is next examined, the L-REF message will be recognized when L.DEQUEUE.MSG is called on it, whereupon PROC.L.REF will be called.

```
(PROC.L.REF
  [LAMBDA (CENTER ID SOURCE)
    (PROG [(WAIT.LIST (A.GETP# C.CENTERS (LIST CENTER (QUOTE WAIT) ID]
          (COND
            ((NULL WAIT.LIST))
            [(EQUAL WAIT.LIST (LIST SOURCE))
              (INIT.C.ACK CENTER ID (CONS CENTER
                                          (A.GETP C.CENTERS
                                                  (LIST CENTER
                                                    (QUOTE LIST) ID]
            (T (A.REMVAL# C.CENTERS (LIST CENTER (QUOTE WAIT) ID)
                          SOURCE])    .
```

The main task of this function is to remove the center that originated the L-REF message from the WAIT list. At the same time, the WAIT list is examined. If it is emptied in the process, INIT.C.ACK is called to generate a C-ACK message and to eliminate the stored information that is no longer needed. The C-ACK message is created by INIT.C.ACK:

```
(INIT.C.ACK
  [LAMBDA (CENTER ID C-LST)
    (PROG [(SOURCE (A.GETP# C.CENTERS (LIST CENTER (QUOTE SOURCE) ID]
          (A.REMPROP# C.CENTERS (LIST CENTER (QUOTE WAIT) ID))
          (A.REMPROP# C.CENTERS (LIST CENTER (QUOTE TEST) ID))
          (A.REMPROP# C.CENTERS (LIST CENTER (QUOTE SOURCE) ID))
          (COND ((EQUAL SOURCE T)
                 (A.REMPROP# C.CENTERS (LIST CENTER (QUOTE LIST) ID))
                 (PRINT.ASSESS CENTER C-LST))
                (T (A.ADDVAL# C.CENTERS
                              (LIST SOURCE (QUOTE L.QUEUE))
                              (LIST ID (CONS (QUOTE CENTER) CENTER)
                                    (QUOTE (TYPE . C-ACK))
                                    (CONS (QUOTE LIST) C-LST])
```

The handling of C-ACK messages is described in the next subsection.

The value of C-LST is the accumulated list of centers that have sent C-ACK messages to the center, or that have been listed in a C-ACK message. It is the accumulated list of centers that the center knows to be operative. When INIT.C.ACK is called for a center, it is given the list that has been stored in that center. INIT.C.ACK accepts this list, adds the center itself to it, and then incorporates this list into the C-ACK message that it is transmitting back down the tree along which the C-ORIG message originally propagated.

The call of PRINT.ASSESS will generally be done only after much processing of C-ACK messages. It is called by INIT.C.ACK when

that function finds that the value of SOURCE stored in the center is T, which indicates that the center is the one that originated the C-ORIG message.

```
(PRINT.ASSESS
  [LAMBDA (CENTER C-LIST)
    (PROG NIL
          [MAPPRINQ ("Network assessment by center #" CENTER
                     " completed." TERPRI (TAB 5)
                     "Reporting centers are: " TERPRI (TAB 10)
                     (MAPSTRING C-LIST) (RPTQ 2 (TERPRI]
          (RETURN (CHARACTER 127])
```

In the simulation, the only thing done with the information returned by the assessment process is to print it out by PRINT.ASSESS. The information would presumably be used in the multiprocessor environment to determine the strategy for recovering from what may be massive damage. The determination and execution of this recovery strategy are not part of the simulation.

k2. C-ACK Responses

When L.DEQUEUE.MSG is called on a local message at a center, it classifies the message by its type. If the type is L-ACK or L-REF, its processing is as has been described. The remaining possibility is that it is C-ACK. If so, PROC.C.ACK is called on it.

```
(PROC.C.ACK
  [LAMBDA (CENTER MSG)
    (PROG [[WAIT.LIST (A.GETP# C.CENTERS
                              (LIST CENTER (QUOTE WAIT) (CAR MSG]
           [SENT.LIST (A.GETP# C.CENTERS
                              (LIST CENTER (QUOTE SENT) (CAR MSG]
           (SOURCE (A.GETP MSG (QUOTE CENTER)))
           (LST (A.GETP MSG (QUOTE LIST)))
           (ID (CAR MSG))
           (OLD.LIST (A.GETP# C.CENTERS (LIST CENTER (QUOTE LIST) ID]
          [COND (OLD.LIST (SETQ LST (APPEND LST OLD.LIST]
          (COND ((NULL WAIT.LIST))
                ((EQUIV.LIST WAIT.LIST SOURCE SENT.LIST)
                  (INIT.C.ACK CENTER ID (CONS CENTER LST)))
                (T (A.ADDPROP# C.CENTERS (LIST CENTER (QUOTE LIST))
                               (CONS ID LST))
                   (A.REMVAL# C.CENTERS (LIST CENTER (QUOTE WAIT) ID)
                              SOURCE])    .
```

This function first saves the list of centers in the message by APPENDing the list onto whatever list already exists. (If none exists, the value of the current list is NIL and APPENDing a list onto NIL simply regenerates the list.)

The function then examines the SENT and WAIT lists. The former should be empty at this time. It may not be so, however, if damage has occurred. What must be ascertained is whether the WAIT list still contains any term that is not in the residue of the SENT list. This is done by EQUIV.LIST:

```
(EQUIV.LIST
  [LAMBDA (LST1 TERM LST2)
    (PROG (FLG)
          [MAPC LST1 (FUNCTION (LAMBDA (ZZ)
            (COND ((OR (EQUAL ZZ TERM) (MEMBER ZZ LST2)))
                  (T (SETQ FLG T]
          (COND (FLG (RETURN NIL))
                (T (RETURN T])    .
```

EQUIV.LIST returns T if LST1 is equivalent (in the set-theoretic sense) to LST2 plus TERM. If this test is passed, PROC.C.ACK calls INIT.C.ACK, described earlier, to generate a new C-ACK message.

This completes the processing of a C-ORIG message and the replies to it. The end product is that the originating center has received a list of all the centers that remain operative and linked to it by paths that involve only operative centers.

k3. Restart

One final detail remains. Let us consider what the situation is after a massive failure. As the C-ORIG message propagates, it may run into various failed centers. Each of these causes a CTR-STATUS message to be initiated and broadcast. As these messages are received, the failed centers are removed from any OK.NBRS list that may contain them. There are, thus, many things happening concurrently. Depending on the order in which events occur, there could be no messages remaining in any QUEUE, L.QUEUE, or IN.PROCESS list, even though not all the required C-ACK messages have been generated.

To avoid this possibility, the final step in EXECUTE.SEND.MSG is to recheck the appropriate lists in each center and, if necessary, restart the processing cycle. The test is done by TEST.NTWK, which calls TEST.CENTER each center in turn.

```
(TEST.NTWK
  [LAMBDA (SIZE)
    (PROG ((N 0) FLG)
      L   (COND ((LESSP N SIZE) (SETQ N (ADD1 N)))
                (T (RETURN FLG)))
          (COND ((TEST.CENTER N) (SETQ FLG T)))
          (GO L])    ,

(TEST.CENTER
  [LAMBDA (CENTER)
    (COND
      ((A.GETP# C.CENTERS (LIST CENTER (QUOTE FAILED))) NIL)
      ((A.GETP# C.CENTERS (LIST CENTER (QUOTE QUEUE))) T)
      ((A.GETP# C.CENTERS (LIST CENTER (QUOTE IN.PROCESS))) T)
      ((A.GETP# C.CENTERS (LIST CENTER (QUOTE L.QUEUE))) T])
```

This final pass through the C.CENTERS list prevents the process from stopping prematurely.

This completes the list of functions used in the simulation, other than those that are a normal part of INTERLISP.

FUNCTION INDEX

Appendix D

CONTROL OF ASSIGNMENTS IN A MULTIPROCESSOR NETWORK*

*Previously published as a Technical Report by Marshall C. Pease under Contract no. F33615-80-C-1014.

I INTRODUCTION

This report discusses algorithms for controlling the assignment of resources in a multiprocessor network. It also documents a simulation program that has been written to test the behavior of a particular algorithm and to demonstrate the approach.

The system concept being explored considers a network of loosely coupled, asynchronous microcomputers communicating through a system of switched connections (i.e., not a bus-linked or multibus system). The application program is assumed to be decomposed into a system of program modules that are nearly autonomous in the sense of Lesser [1]. It is assumed that each program module is at least small enough to be run in a single microcomputer, and perhaps even small enough to allow several to be executed in the same microcomputer. The assumption of near-autonomy implies that each module contains most of the information it needs for its own operations and that only occasionally is intermodule communication needed for coordination. Therefore, communications can be handled by a message-passing routine, allowing the use of a loosely coupled system.

The design is intended to ensure a very high degree of fault-tolerance and survivability. It is also intended to provide the adaptability that will allow the system to remain operational as the demands of the application program change with time and from one installation to another. For example, if the program is transported to a system with a different hardware configuration, perhaps using different numbers of microcomputers, it should not be necessary to reprogram.

We can summarize the requirements in the diagram of Figure 1.

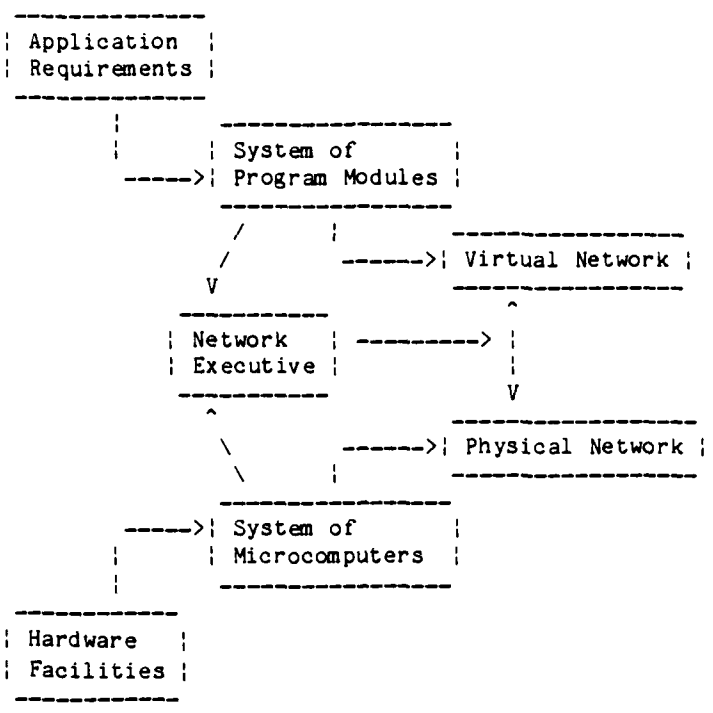


FIGURE 1 CONCEPTUAL OUTLINE OF SYSTEM

The application program sees only the virtual network implied by its intermodule communications. This virtual network is independent of the hardware and is unaffected by any change in the hardware that may be the result of failures or damage or of a move to a new system. To achieve this independence, control of the assignment of program modules to microprocessors is the responsibility of the network executive. This unit must operate autonomously, independently of the details of the application program, but using its own knowledge of the requirements of the various program modules. It is the network executive that determines when the assignment map must be changed and how--and that executes the change. Furthermore, the network executive must be able to exercise this control dynamically, with as little interruption of operations as possible.

To make the program independent of the physical network, the network executive must manage much more than the assignment of program modules to microcomputers. For example, it must manage communications so that they are not disrupted by any changes in the assignment map. An algorithm for this function and a simulation program to test the algorithm have been described in a separate report [2]. Other areas such as the detection and diagnosis of faults, also require treatment. In the work being reported here, however, we are concerned only with dynamic control of the assignment map.

The major assumptions being made about the hardware are the following:

- The network is assumed to be a switched, lattice type one. By this we mean that each processor is connected directly to a restricted set of neighbors through a set of distinct links. This is in contrast to a network that uses bus lines to connect many centers.

- The links between neighbors are bilateral. Therefore, if center i is a neighbor of j, j is a neighbor of i.

- The total number of centers in the network is limited, but is significantly larger than the number of neighbors of any center. Typically, we expect a network to contain between, say, 16 and a few hundred centers, and each center to have, for example, four to eight neighbors.

- The connectivity of the network is high, so that there are many alternative paths between pairs of centers. The network should not become disconnected easily as faults accumulate.

The assumptions we make about the application program are as follows:

- The program is assumed to be composed of a number of nearly autonomous modules each of which is small enough to be executable in a single processing center. (A center, however, may be able to handle several modules concurrently.)

- Each module contains most of the values and data it needs for its own operations. Transfers of information between modules will be needed for coordination, but this need can be met satisfactorily by a message-passing system.

- The individual program modules, or most of them, are nonterminating. For example, the function of a module may be to maintain information about the current state of some entity in the real world. The module will need to update its information periodically, without termination.

- It is assumed that the programmer can make useful estimates of the load created by each module. These estimates can be used to predict what combinations of modules will not overload a center.

- It is assumed that priorities exist that define graceful degradation when necessary. In addition, it is assumed that the higher-priority modules are programmed to allow their continuing operation in a degraded mode. For example, a high-priority module may have to substitute estimations for the more exact information that would normally be available. If so, estimation procedures are included in the module.

- It is assumed that it is possible to reconstruct the current state of a module from checkpoint data describing a previous state. This implies that the detailed behavior of a module is determined by a limited set of parameters. The purpose is to ensure that the module can be reconstructed from a restricted amount of data and information, i.e., the checkpoint data and the parameters.

These assumptions about the program are applicable when we can use a programming methodology such as that developed and described elsewhere [3, 4] using M-modules. The suitability of the methodology depends on the nature of the application requirements, but there is a broad class of important applications for which the method is applicable and useful.

It is the final assumption about the program that allows the dynamic adjustment of assignments. In effect, it lets modules be "relocated" as necessary, although they are not actually relocated; instead they are recreated at a new location. A backup for a module being executed in one center is maintained in another. This backup includes the parameters that are needed, plus checkpoint data for a recent epoch. Should it become necessary to reassign the module, the backup is activated. The backup's state is updated and a new backup created in a different center. The original module can then be abandoned. What was previously the backup is now recognized by the network executive as the main copy and, after activation, is the one used for all coordination purposes.

The listed assumptions make feasible the system concept outlined in Fig. 1. The nature of the program modules determines the intermodule communications that are needed and that generate the virtual network. If the network executive can translate these requirements into operations in the physical network, the application program can be made entirely independent of the latter.

In this report, we shall examine in detail and under the listed assumptions an algorithm to control the assignment of modules to processing centers. As

indicated, the dynamic control of resource assignments is central to the system concept, although it is far from being the only important element. It is central, however, in that the other elements of the network executive must be able to operate in the fluid environment created by the assignment algorithm. The algorithm developed here has been implemented in a simulation program, the details of which are described in the appendix.

II REQUIREMENTS

An algorithm for control of the locations of a set of modules should meet several requirements. These can be summarized as follows:

- Controlled by Environment. By the environment we mean the identities and properties of the available resources, and the demands made on those resources by the application program. The environment is regarded as fixed unless there is a change in either the available resources or in the load imposed by one or more modules.

- Locally Deterministic. The algorithm is deterministic, at least locally within a component. We may permit control to pass between the different centers in a nondeterministic way.

- Stable. The algorithm must always lead to a stable state from which no further changes occur while the environment itself remains unchanged. It is not required that this state be unique for a given environment, only that some stable state be reached from any initial state.

- Distributed. The procedures that implement the algorithm should be replicated as necessary and copies distributed across the network. Each component so generated should act autonomously according to its own information and procedures.

- Locally Controlled and Locally Effective. The actions of the component in a processor should depend only on the state of that processor and the states of the processor's neighbors in the network. To the extent possible, the component's actions should directly affect only the processor and its immediate neighbors.

The first requirement essentially defines the environment. There is an implied assumption that the environment does not change too rapidly; at least we should be able to consider the environment as essentially constant during the time it takes to reach the final stable state from any initial state.

That the algorithm should be locally deterministic may not be essential, but there seems no reason to look for a more general approach. A locally deterministic algorithm is likely to respond more rapidly than a

nondeterministic one.

The third requirement, for stability, is critical. We must be able to demonstrate that the system will always reach some state that will no longer be modified by the algorithm. Since the number of possible states is finite, stability is assured if no state is repeated. However, this still permits a long sequence of changes before stability is reached.

The importance of a distributed network executive follows from the consideration that a centralized control system is vulnerable both to faults within the control unit itself and to failures of communication to and from the control unit. It may not be necessary for every processor to have its own component, but it appears that it is necessary to distribute the control system sufficiently widely to limit its vulnerability.

The final requirement is that the algorithm be locally controlled and locally effective. We consider these requirements essential for a fault-tolerant, multiprocessor system. We believe that each component of a distributed network executive should use only locally available information and initiate actions only within the immediate neighborhood of its host processor.

The reason for the restriction to locally available information is to prevent making the system vulnerable to failures in the communication processes. If global information were used, we could not be certain that all parts of the network have the same information at any given moment. If different parts of the network have conflicting information, they can reach conflicting conclusions about what actions to take. This could lead to inconsistent actions in different parts of the network, possibly leading to deadlock or other disastrous condition.

Similar considerations argue the desirability of limiting the range of immediate influence of a network executive component. If a command had to be disseminated widely, elaborate precautions would be necessary to ensure that the dissemination is done correctly--without errors and without failing to reach any center that should be affected. Otherwise, as in the previous case, different parts of the network could be made to act in conflict with one another.

The requirements for local control and local effectiveness are intended to limit the range of influence of any component of the network executive. Without such a restriction, we believe it would be almost impossible to ensure the continued coordinated of the network in the presence of faults or other event: requiring adaptation of the network's behavior.

There are two other properties of an assignment algorithm that are important:

- Rate of Convergence. How can we measure the rate of convergence to the final stable state, and can we specify a minimum rate that will apply under all conditions?

- Bound on the Convergence. Can we specify an upper bound on the number of assignment changes that can occur before a stable state is obtained?

In the analogous problem for continuous systems, we would want to know, if possible, that the final stable state is always reached in a finite length of time. We would also like to be able to specify an upper bound on the time necessary to reach the stable state. The discrete nature of the problem being studied ensures both finiteness and boundedness. There are only a finite number of states possible. If stability is reached at all in a deterministic system, it must be reached in a finite number of steps, and a bound exists on what this number can be for any initial state.

Although we can be sure of finiteness and boundedness, we still would like to have useful quantitative limits on the rate of convergence and on the number of steps that may be necessary in the worst case. We would like to have a measure of the rate at which the system approaches stability, and a useful lower limit on this rate. We would also like to have a measure of the distance between states that we can use to obtain a useful upper bound on how far from the final stable state the system can be. These measures will be helpful in evaluating the effectiveness of a given algorithm.

III GLOBAL AND LOCAL MEASURES

One of the interesting results of this investigation has been the discovery that, under appropriate conditions, a global measure can be used to derive a distributed control algorithm with the desired properties. In this section, the applicable analytic techniques are discussed.

We start by defining a global measure and using it to prove conditions under which the system will behave as required. That is, the global measure is required to prove stability, to establish a lower bound on the rate of convergence and also, when possible, an upper bound on the amount of action needed before stability is achieved. These are global properties in that the behavior being measured is concerned with the state of the network as a whole.

By a measure we mean a function that assigns a value to every possible state. The value can be numeric, Boolean, or other (e.g., OK or NG, or good, better, best). If numeric, it can be integral, rational, or otherwise. The only requirement is that the measure should provide a partial ordering among the set of possible actions.[1] This ordering determines what actions are permitted by the algorithm and contributes to the choice of the action to be executed.

A measure is local if its value is determined entirely by the states of a processor and its immediate neighbors. The measure is global if its value is determined by the state of the entire network. With a locally controlled algorithm, each component uses a local measure to determine which actions are

---

[1] An ordering relation on a set S of elements, a, b, c, ..., is a binary relation r such that, if r(a, b) and r(b, c) hold, then r(a, c) holds (transitivity), and if r(a, b) holds, then r(b, a) does not and vice versa (antisymmetry). S is completely ordered by r if, for any pair of distinct elements, x and y, of S, either r(x, y) or r(y, x) holds. Otherwise the orderirg is partial.

permitted and to choose among permitted actions. The algorithm can be regarded as defined by the local measure.

A global measure is one whose value is determined by the state of the entire network. A suitable global measure can be used to evaluate the overall effect of the algorithm. In particular, the requirement for stability is a global property. Both the lower bound on the rate of convergence and the upper bound on the amount of change necessary to reach a stable state are global aspects of the process--and a global measure can often be used to test an algorithm. The method is closely related to Liapunov's direct method for the study of stability in dynamic systems. [5] For this purpose we require a global measure with the following properties:

1. The measure provides a partial ordering among all possible states of the system.

2. A boundary can be specified such that the value of the measure is always on a specified side of the boundary. As a rule, the measure is nonnegative real and the boundary is zero.

3. To ensure stability, all changes permitted by the algorithm must bring the measure closer to the boundary. It is assumed that, if one or more changes are permitted, one of them will be executed; the system is stable if and only if there remain no permitted actions to be executed. Since the value of the measure approaches the boundary but cannot cross it, any sequence of permitted changes must tend towards a state from which no further changes are permitted.

4. A lower bound on the rate of convergence is determined if the value of the measure always changes by at least some specified amount.

5. An upper bound is established for the number of changes that may be needed to reach stability if an upper bound can be given for the value of the measure in any given environment.

A measure satisfying (1) and (2) is always isomorphic to one with nonnegative real values. For example, if the original measure is nonpositive, its sign can be changed. If the values lie in some arbitrary field or ring (for example, the values have complex values) and are ordered by their positions along some defined trajectory, the distance along this trajectory can be used as an equivalent nonnegative, real measure. It is always possible, also, to transform the measure so that the boundary is zero. This is not always convenient to do, however, since it may make the measure dependent on the environment. But it is convenient to require that the boundary be either zero

or a positive real number. Henceforth, we will assume that such a transformation of the measure has been made if necessary.

We now assume the existence of a global measure M that can be expressed as the sum of a set of values, W(i), where each W(i) depends only on the state of processor i. This assumes that the measure is separable, with each center making its own contribution to the total value. Furthermore, the separate contributions of the processing centers are taken as additive. Although these restrictions are much tighter than necessary, they do not interfere with our immediate goal. We will, therefore, leave the problem of finding the most general, nonrestrictive conditions for later study.

Consider now the set S of all possible elementary actions. By an elementary action we mean one that is not regarded as composed of any simpler actions. The algorithm will decide either to execute an elementary action, or not, and will not consider executing a part of an elementary action.

The actions in S are not limited to those that may be possible at a given time. They include all elementary actions that are possible starting from any possible state of the system.

The actions of S must be able to provide a basis for any possible state change. That is, any state change must be realizable as the result of executing a sequence, generally not unique, of members of S.

To illustrate, the actions of S can be specified as all those that transfer a named program module from one particular processing center to a named neighbor. An action in S is not executable unless the named program module is in the named processing center, but the action is still listed in S.

Let s(i, j) be any member of S where i is distinct from j. For example, s(i, j) may shift a specified module from center i to center j. We assume that this action changes only the values of W(i) and W(j) of M. This implies that every executable action affects exactly two components of M. This is not necessarily true, we could specify elementary actions that shift more than one module and involve more than two centers. Also, the shift of a module from i to j could affect a third center in some way. The additional complications that arise when we include these possibilities do not seem significant. For the present,

at least, we ignore the possibilities.

We can also observe that the definition of S implies a specification of the neighborhood of a center. If s(i, j) is in S for some i and j, then centers i and j are neighbors of each other.

The next step is to partition[2] S into a set of subsets, S(i). Each S(i) will specify the set of possible actions that are to be controlled by the ith component of the assignment algorithm. The partition is constructed so that every member of S(i) is either an s(i, j) or an s(j, i) for some j. That is, if an action is controlled by the component resident in the ith processor, one of the components of M affected by that action is W(i). Since the set of all S(i) is a partition of S, every action of the form s(i, j) is assigned either to S(i) or to S(j), but not both.

The definition of the S(i) components of the partition ensures that, if the algorithm is such that its ith component controls only the actions in S(i), the algorithm will be locally effective. We have observed that the fact that an s(i, j) is in S implies that centers i and j are neighbors. Hence, if the ith component of the network executive controls the set S(i) of possible actions, the algorithm is, by definition, locally effective.

The global measure M evaluates the effect of any action in S. Every elementary executable action results in a change in the value of M. The action is permitted only if it decreases the value of M by at least some minimum amount. We can obtain a local measure with which to control the ith component of the algorithm simply by restricting M to S(i). This measure is not locally controlled, since it depends on the state of the whole network, but it does allow the algorithm to be distributed. Furthermore, it is easy to modify the algorithm to make it locally controlled as well as locally effective.

To achieve local control, we recall that any action in S(i) affects only the two components of M, W(i) and W(j), where j is some neighbor of i. Providing the proposed action is taken in isolation, independently of any other action

---

[2]A partition of a set S is a set of subsets such that every member of S is in one but only one of the subsets.

in the network, the resultant change in M is due entirely to the changes in W(i) and W(j). We can therefore define a local measure, M(i), as the sum of W(i) plus the W(j) for all j that are neighbors of i. As long as there are no changes in other parts of the network, M(i) differs from M by a constant value. If M(i) is used to control the ith component of the algorithm, the algorithm will be locally controlled.

It is an important assumption that each elementary action, s(i, j), is considered in isolation. In practice the condition can be somewhat relaxed, but it cannot be removed entirely. The problem is that center i might decide that s(i, j) should be done at the same time that center k decides on s(k, j). Both actions affect W(j). In the global context, it is possible that either action would improve M, but not both together. Executing either one can make the other impermissible.

To avoid this kind of interaction, we need a mechanism to ensure that, while center i is executing its component of the network executive, center j is prevented from executing its component, where j is any neighbor of i or any neighbor's neighbor. Given a coordinating procedure that enforces this restriction, the remaining possibilities of overlapping actions cannot lead to conflict.

Finally, we can note that the possible values and properties of M(i) limit the global behavior. In particular, if the change in any M(i) for any permitted action is at least epsilon, the same is true of M. Therefore epsilon is a lower bound on the rate of convergence of the system from any initial state. Also, by examining the values of the M(i) for various initial states, and the ways in which they can combine into M, it is often feasible to find a maximum value for M, given a particular environment. This maximum value, with the minimum rate of convergence, leads immediately to a bound on the number of actions needed to reach stability from any initial state.

This completes the general development. We have shown that it is possible to use a global measure to derive algorithms that are distributed and that have the desired global properties. The resultant algorithms are not unique. There are many opportunities for variation. For example, the definitions of the W(i) can be varied without changing M. The determination of what actions are elementary and so included in S is also subject to variation. Finally,

there is considerable variation possible in the way S is partitioned. The effect of these possible variations on the resultant algorithm have not yet been examined. The immediate purpose, to show that a suitable algorithm can be developed, has been accomplished.

IV REPRESENTATIVE ALGORITHM

In the previous section, we developed a technique that allows the generation of assignment algorithms that satisfy the stated requirements. Different algorithms will result, depending on the precise definition of M and of its components W(i), the specification of the actions included in S, and the way S is partitioned into the S(i). The simulation program detailed in the appendix, however, is described in terms of a specific algorithm. While the simulation program can be quickly adjusted to reflect different algorithms, the existing program uses what we shall the basic algorithm.

The basic algorithm uses a real, positive number that is specified by the user for each module. This number is recorded as the value of the property RQMT for the module. This value is considered to be an estimate of the processing power, measured in whatever terms are appropriate, that is needed by the module. It is assumed that the values of RQMT of separate modules can be combined additively, so that the total load on a processor is the sum of these values for all the modules being executed in the processor or held in the module in inactive status.

The global measure M that is used is the sum of the squares of loads in all processors. This measure is of course nonnegative. Furthermore, it is bounded for any specified set of program modules. The worst case obtains when all modules are in a single processor. In that situation, the value of M is the square of the total load of the entire application program. If the sum of the values of RQMT for all the program modules is L, the maximum value of M is L^2.

This measure M is nonnegative but cannot reach zero in any environment in which L is nonzero. A better value for the boundary corresponds to the case in which the total load is distributed uniformly across the network. Of

course, the discrete nature of the program modules may prevent this limit from being reached, but it is certainly a more realistic boundary than zero. If there are n processors in the network and if the total load could be distributed evenly among them, each processor would see a load of (L/n). The resultant value of M would be n(L/n)^2 or (L^2)/n. Therefore, the value of M lies in the range:

$$(L^2)/n \leq M \leq L^2 .$$

This is still not the exact lower bound if the modules cannot be distributed exactly evenly. Fortunately we do not need an exact lower limit; most of what we need is simply the result of the existence of a boundary. Knowledge of an approximate value is quite adequate for the rest.

We have commented that, for any measure, it is always possible to make its boundary zero by subtracting the value of a nonzero boundary from the measure. This is true, but would be inappropriate here, since it would make the definition of the measure dependent on the environment which includes the set of program modules and the values of RQMT specified for each. Furthermore, it is not difficult to use a nonzero lower bound.

The elementary actions that, collectively, form the set S are those that move a specified module from one center in the network to any neighboring center. No additional restrictions are included in the basic algorithm.

In the basic algorithm, the partition of the set S of elementary actions is such as to make the algorithm a "push" one. That is, each processor, when executing its component of the assignment algorithm, determines if it would be useful to transfer any of its modules to a neighbor. In a "pull" algorithm, a center would consider if it should acquire a module from a neighbor. A mixed strategy using both push and pull approaches is also possible.

Finally, the basic algorithm actually controls the location of the backup modules, rather than the modules themselves. The reason is that this seems to correspond most closely to what is actually needed. The backups are there to provide for a quick recovery from a failure. Once a failure is detected, the backup can be brought on line as soon as it can be updated. No communications are needed after the order to activate the backup. Therefore, the critical decision is where to place the backup, keeping the location of the module

fixed. In effect, this approach assumes that the distribution of the modules remains satisfactory as long as there is no failure or any other event that triggers one or more of the backup modules.

This is the algorithm implemented in the simulation program described in the appendix. A sample of its operations is given in the next section.

V SAMPLE RUN

In this section, we give an example of a run of the simulation program. The network used for this run is a (2 X 2 X 2) cube with processing centers labeled 1 through 8 at each node. All processing centers are presumed equivalent, each having initially a capacity of 10 on an arbitrary scale.

The run uses 18 program modules labelled A through S. Each module has been assigned a number in the range from 1 through 5 as the value of the property RQMT. This number represents the load the module imposes on a center in the same units as the center's capacity. The modules have also been assigned priorities from 1 to 10, higher numbers indicating higher priorities. The priorities are used to determine which modules are to be sacrificed if graceful degradation becomes necessary.

The network is loaded through center 1. All modules are first placed in that center in inactive status. The network executive then distributes the modules around the network so as to obtain as good a balance as possible of the potential load. Next a backup for each module is created in a center that is a neighbor of the one containing the module, the choice of neighbor being made so as to distribute the potential load. Finally, modules are moved into active status, simulating actual operations. The activation of the modules in a center is done in the order of decreasing priority. In the simulation, the total load at each center does not exceed the center's capacity, and all modules become active. The state of the network at this point is summarized in Figure 2.

Figure 2 is a copy of a computer printout that is intended to display the state of the network in a convenient format. The different columns show various information about each center in turn. The next to final column, labeled Total load, is the sum of the values of RQMT for every module in

active status in the center. The final column, labeled Total Rqmts., is sum of the values of RQMT for every module in the center, whether in active or inactive status.

Center	Capacity	Active Modules	Inactive Modules	Backup Modules	Total Load	Total Rqmts
1	10	R,M,P		G,I	6	6
2	10	G,J,K		P,B	7	7
3	10	Q,H		M,R,C	6	6
4	10	F		K,H,D	5	5
5	10	A,O,I		L	9	9
6	10	L,B		J,A,N	6	6
7	10	E,C		Q,O	6	6
8	10	N,D		F,E	6	6

FIGURE 2 STATE OF THE NETWORK AFTER INITIAL LOADING

A few of the actions that lead to this state are shown in Figure 3 which is a reduced copy of a computer printout of the history list that records the sequence of actions. The length of the full list is indicated by the numbers in the first column.

No.	Action	Module	Old Center	New Center
Modules Loaded into Center #1				
1	Shift Inactive Mod	F	1	2
2	Shift Inactive Mod	C	1	3
	. . .			
13	Shift Inactive Mod	F	2	4
	. . .			
24	Shift Inactive Mod	J	1	2
Modules Distributed, Execution Not Yet Started.				
25	Backup Created	M	3	n/a.
26	Backup Created	P	2	n/a.
	. . .			
42	Backup Created	N	6	n/a.
Backups Set.				
43	Activate Module	R	1	n/a.
44	Activate Module	M	1	n/a.
	. . .			
60	Activate Module	D	8	n/a.
Execution Started, as Limited by Capacity.				

FIGURE 3 INITIAL LOADING AND DISTRIBUTION SEQUENCE

As shown in Fig. 3, it took a total of 60 distinct steps to reach the state shown in Fig. 2. The first twenty-four events shift modules about. For example, module F is first moved from center 1 to 2 in event #1, then to center 4 in event #13. Events #25 to #42, inclusive, are those that create the backup versions for each of the 18 modules. Events #43 to #60, inclusive, are those that simulate the start of execution of the modules.

After initialization was completed, a command was entered that simulated the

discovery that center 2 had failed. This command not only reduced 2's capacity to zero; it caused the system to rebalance its load to adjust to the failure event. The state that resulted is shown in Figure 4.

Center	Capacity	Active Modules	Inactive Modules	Backup Modules	Total Load	Total Rqmts.
1	10	R,M,P,G		I	8	8
2	--					
3	10	Q,H		M,R,C,K	6	6
4	10	F,K		H,D	8	8
5	10	A,O,I		L,G,P	9	9
6	10	L,B,J		A,N	8	8
7	10	E,C		Q,O	6	6
8	10	N,D		F,E,J,B	6	6

FIGURE 4 STATE AFTER PROCESSOR 2 FAILS

Before the failure, center 2 was executing modules G, J and K. The backups copies of these modules were in processors 1, 6 and 4, respectively. These backups have now been activated, and new backups for these modules created in centers 5, 8 and 3, respectively. In addition, center 2 had the backup copies of modules P and B being executed in centers 1 and 6, respectively. New backup copies of these modules have been created in centers 5 and 8, the centers being chosen in accordance with the assignment algorithm.

Next, a command is entered that simulates the discovery that center 5 has failed. Again, there is considerable adjustment. The final state of the network after a stable state is reached is shown in Figure 5.

Center	Capacity	Active Modules	Inactive Modules	Backup Modules	Total Load	Total Rqmts.
1	10	P,G,I,R		M	9	9
2	—					
3	10	Q,H,M		R,C,K,O,I,G,P	8	8
4	10	F,K		H,D	8	8
5	—					
6	10	B,J,A		N,L	7	7
7	10	E,C,O		Q	9	9
8	10	N,D,L		F,E,J,B,A	10	10

FIGURE 5 STATE AFTER PROCESSOR 5 FAILS

It is interesting to observe that, aside from the shifts that are immediately required by the failure, module M has been shifted from 1 to 3 and module L from 6 to 8. Both of these shifts are the result of load balancing operations.

The example illustrates the way the network responds to events that signal the failure of processing centers. Provided only that the network has not been disconnected by a particularly unfortunate sequence of failures, the capacity for continued operation is maintained.

VI MASSIVE FAILURE OR DAMAGE

Special consideration is necessary when the system must recover from an event that causes the simultaneous failure of many centers. Such an event may result from physical damage that renders a substantial fraction of the network inoperative.

The techniques illustrated in the last section are not adequate when massive damage has occurred. There is danger that the damage may have destroyed not only the center executing a module, but also the center holding its backup copy. The backup is located in a different center from that holding the module being executed, but, in the case of massive damage, both centers can be destroyed simultaneously. It is also quite possible that the network may have been cut into separate fragments. If this should happen, we would still want to maintain as much operational capability as possible, but this means that operations would have to be confined to one of the surviving fragments.

We could minimize the chance of losing modules by making several copies available. In the extreme case, we might even create a backup for every module in every center. This would still not protect against possible fragmentation of the network.

The simplest approach seems to be to depend on the initial loading algorithm. That is, if a massive failure occurs, the execution of any remaining modules is suspended and the system is reloaded and restarted with archived checkpoint data. If the reloading is done through a single center, as in the simulation, the initial distribution of the modules cannot shift any of them into a disconnected part of the network. If the network has been fragmented, the separated parts will simply fail to receive any modules and will remain quiescent.

A further advantage of reloading is that it causes the highest-priority modules to be the first ones returned to operation. If sufficient capacity remains so that degradation is not necessary, the most critical modules are the first returned to operation. If insufficient capacity remains to maintain the full application program, the procedure will result in the automatic degradation of operations to the extent necessary.

The requirements for recovery from massive failure or damage are therefore met by the procedure that initially loads the system and brings it into operation.

VII CONCLUSIONS

In this report we have discussed in considerable detail the requirements for an algorithm to manage the dynamic assignment of program modules to the processing centers of a multiprocessor network. We have shown that it is possible to obtain an algorithm that can not only fulfill these requirements, but can provide the on-line responsiveness necessary to achieve a high degree of fault-tolerance and survivability.

An important outcome of this work has been the recognition of the possibility of deriving local algorithms from a global measure that ensures the desired properties. The global measure can be chosen to have the properties of a Liapunov function and be used to ensure stability; it can also provide a lower bound on the rate of convergence and an upper bound on the number of steps needed to achieve a stable condition. If suitably constructed, the global measure can be decomposed in a way that results in a distributed, locally controlled, locally effective algorithm that will enforce the global properties.

There is a class of global measures that can be decomposed into local algorithms. The decomposition of the total measure M into the W(i) associated with each processing center can be varied. The variations will cause the matching algorithms to emphasize different aspects of the distribution.

The procedure for deriving the assignment algorithm from the global measure can also be varied in number of ways. The partition of S into the various S(i) has a degree of arbitrariness to it. We have noted the possibility of deriving either a push or a pull algorithm, or one that is a mixture of the two. In addition, we could impose restrictions on certain of the s(i, j), setting up the algorithm so that certain modules will be avoided by some centers, sought by others.

Another class of variations can be introduced by restricting the set S. By limiting S, the transfer of certain modules can be limited to some subset of the network's processors. This device would allow consideration of inhomogeneous networks in which some processors have quite different operational characteristics than do others.

Another problem that might be addressed by a suitable modification of the technique is to develop an assignment algorithm that will maintain clusters of certain modules. The objective is to execute all modules in a given set in processing centers that are near one another. The set of centers in which a set of modules is being executed may change as failures occur. The needed adjustments should be chosen in a way that will preserve the clustering properties.

It seems likely that a cluster-preserving algorithm can be developed by limiting the set of executable actions S to those which do not upset the cluster relationship. The precise definition of the clustering properties that need to be preserved would have to be carefully formulated. It would probably also be necessary to introduce constraints on the S(i), or their evaluations, that are functions of the conditions present in the neighborhood of processor i. The process will need careful study to ensure that the relation between the local algorithm and the global measure is preserved. However, the approach appears to be a potentially powerful one that should be examined.

There are therefore a number of interesting extensions that would warrant deeper investigation. The most important results for immediate purposes is that we have found an algorithm with the required properties that fits our immediate needs.

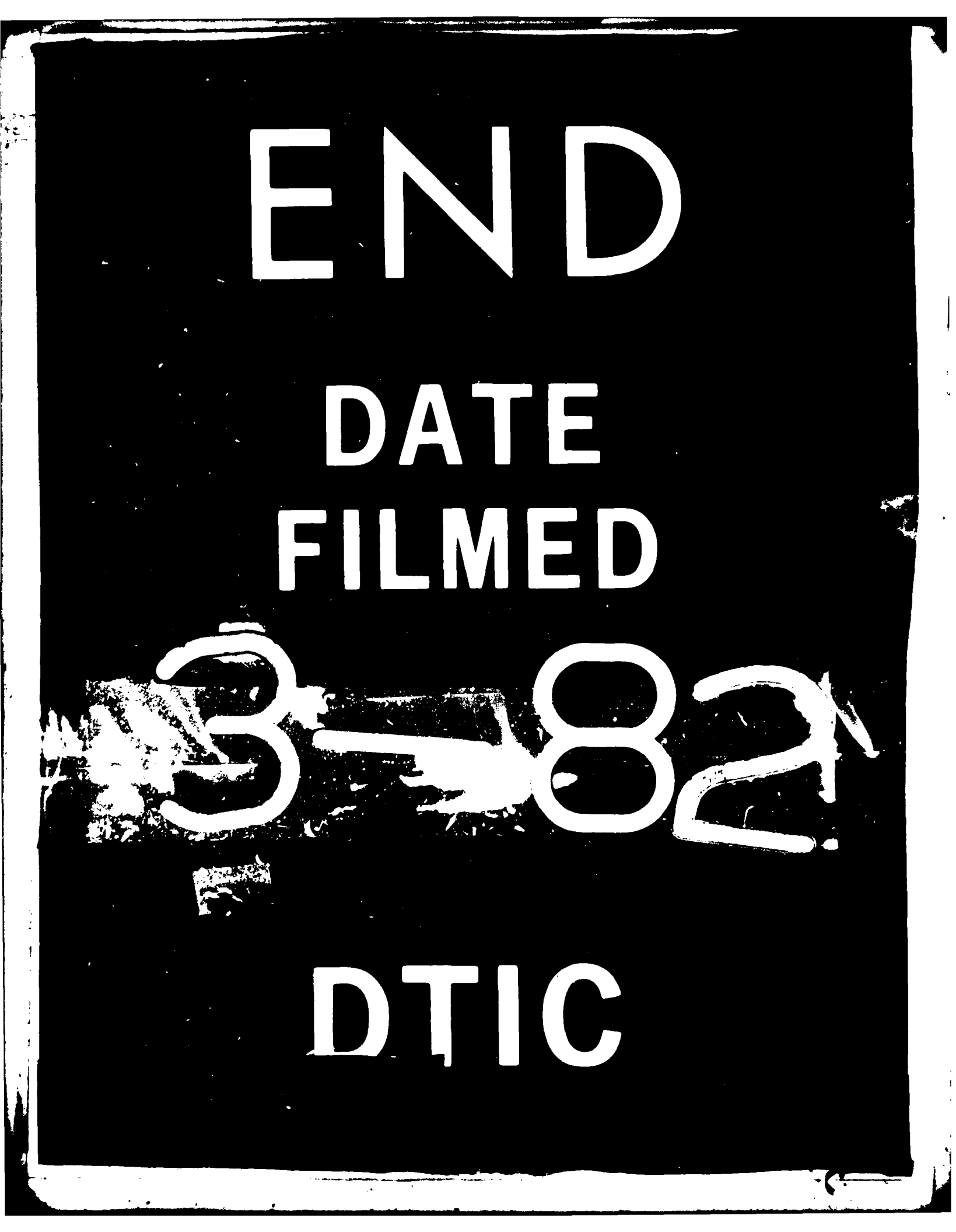
REFERENCES

1. V. R. Lesser and D. D. Corkhill, "Functionally-Accurate Cooperative Distributive Systems," COINS Technical Report 79-12, University of Massachusetts, amherst, Massachusetts (February 1979). Also in "Working Papers in Distributed ComputAtion: Cooperative Problem Solving," Distributed Computation Group, Computer and Information Science, University of Massachusetts, Amherst, Massachusetts (July 1979).

2. M. Pease "Control of Communication in a Multiprocessor Network," Technical Report 1, Contract F33615-80-C-1014, SRI Project 1314, SRI International, Menlo Park, California (June 1980). 3. M. C. Pease "M-Modules: A Design Methodology," Technical Report 17, Contract No. N00014-77-C-0308, SRI Project 6289, SRI International, Menlo Park, California (March 1979).

4. C. A. Monson, P. R. Monson, and M. C. Pease "A Cooperative Highly-Available Multi-Processor Architecture: CHAMP," Proceedings of COMPCON 1979, Washington, D.C., pp 349-356 (September 1979).

5. J. La Salle and S. Lefschetz, "Stability by Liapunov's Direct Method with Applications," Academic Press, New York, New York, 1961.


END
DATE
FILMED
3-82
DTIC